

Best Available Copy

⑫ DEMANDE DE BREVET D'INVENTION

A1

②2 Date de dépôt : 25.09.90.

③0 Priorité :

④3 Date de la mise à disposition du public de la demande : 27.03.92 Bulletin 92/13.

⑤6 Liste des documents cités dans le rapport de recherche : *Se reporter à la fin du présent fascicule.*

⑥0 Références à d'autres documents nationaux apparentés :

⑦1 Demandeur(s) : Société dite GEMPLUS CARD INTERNATIONAL Société Anonyme — FR.

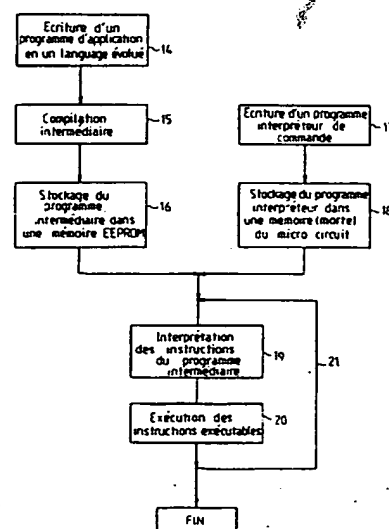
⑦2 Inventeur(s) : Gordons Edouard, Grimonprez Georges et Paradinas Pierre.

⑦3 Titulaire(s) :

⑦4 Mandataire : Cabinet Ballot-Schmit.

⑤4 Support portable à micro-circuit facilement programmable et procédé de programmation de ce micro-circuit.

⑤7 Pour résoudre des problèmes de programmation dans des cartes à puce, on munit le micro-circuit contenu dans ces cartes d'un programme interpréteur de commande. De plus on compile des programmes d'application en un langage intermédiaire compréhensible par l'interpréteur de commande. On montre que non seulement on gagne de la place en mémoire, mais en plus on facilite la tâche des programmeurs qui n'ont plus alors qu'à programmer leurs applications en un langage évolué qui a priori n'était pas adapté à la programmation des cartes à microprocesseur.



FR 2 667 171 - A1



A

SUPPORT PORTABLE A MICRO-CIRCUIT FACILEMENT PROGRAMMABLE
ET PROCEDE DE PROGRAMMATION DE CE MICRO-CIRCUIT

La présente invention a été faite en collaboration avec le LABORATOIRE D'INFORMATIQUE FONDAMENTALE DE LILLE et avec le CENTRE D'ETUDES ET DE RECHERCHES EN INFORMATIQUE MEDICALE dépendant respectivement de l'Université de Sciences et Techniques de Lille et de l'Université de Droit et de la Santé de Lille. La présente invention a pour objet un support à micro-circuit dont le micro-circuit est facilement programmable ainsi qu'un procédé de programmation de ce micro-circuit. Elle trouve plus particulièrement son application dans le domaine dit des cartes à puces. Dans ce cas le support est une carte au format carte de crédit. L'invention a pour objet de mettre à la disposition des programmeurs, pour des applications diverses devant avoir un caractère portable, la puissance de travail procurée par de tels supports. Par caractère portable on entend le fait que la carte, ou plus généralement un support quelconque de faible taille (quelques centimètres) et de faible poids (quelques centaines de grammes), peut en étant insérée dans un lecteur établir une transaction entre une machine reliée à ce lecteur et la carte. Cette transaction s'effectue alors selon un protocole et selon des instructions qui sont contenues dans la carte.

La principale difficulté rencontrée avec les cartes à micro-circuit vient de ce que le microprocesseur dont elles sont pourvues est associé à une mémoire de travail (de type RAM, statique ou dynamique) de faible capacité, quelques fois seulement 128 octets, et à des mémoires programmes (ROM: la plupart du temps de type

EPROM ou même EEPROM) elles aussi de très faible capacité, généralement limitée à quelques dizaines de kilooctets voire seulement à quelques kilooctets. De plus, les microprocesseurs des cartes sont généralement
5 des microprocesseurs avec chacun un jeu d'instructions réduit. Ces microprocesseur sont même quelquefois assimilables à des micro-contrôleurs dont les bus d'échange ne sont pas en totalité accessibles depuis l'environnement extérieur du micro-circuit. La variété
10 de conception de l'organe essentiel de traitement de ces microprocesseurs pour carte à micro-circuit a conduit à ce qu'il existe sur le marché un nombre important de microprocesseurs différents. On notera que ceci n'est pas le cas pour les gros microprocesseurs, avec jeu
15 d'instructions important, dont la complexité a conduit à un nombre très limité de familles du fait du faible nombre des entreprises capables de les fabriquer.

Dans le domaine des cartes à micro-circuit le programme représentatif de l'application à mettre en
20 oeuvre est généralement stocké dans une mémoire morte non volatile contenue dans le micro-circuit.

Cette diversité des microprocesseurs pour carte induit, pour le programmeur qui est désireux d'utiliser une carte à micro-circuit, la nécessité de connaître
25 parfaitement le langage machine du microprocesseur utilisé. Ceci n'est pas possible si on veut pouvoir utiliser plusieurs types de microprocesseur différents. En outre, du fait du nombre limité d'instructions exécutables par les microprocesseur des cartes à
30 micro-circuit, les langages de programmation dits évolués peuvent ne pas être complètement utilisables, il convient de les tester. Parmi ces langages évolués on fera référence, à titre d'exemple, aux langages dits C, COBOL, PASCAL, BASIC, ADA, ainsi qu'à de nombreux autres

également connus.

On rappelle que la programmation par un programmeur, une personne humaine, d'un programme informatique est facilitée par l'utilisation de ces langages évolués. En effet ces langages sont à la fois proches du langage parlé (les instructions sont claires: par exemple WRITE, IF, GO TO...), et puissants parce que chaque instruction en langage évolué est ainsi exprimée en raccourci (instruction donc non exécutable telle quelle par le microprocesseur de la carte), et qu'elle peut être transformée, automatiquement par la suite, en une série d'instructions en un langage machine, lui, compréhensible et exécutable par le microprocesseur. Cette transformation s'effectue en une opération ultérieure à la programmation et appelée COMPILATION. Si le langage n'est pas un langage évolué, mais uniquement un langage mnémonique dit assembleur, proche de celui du microprocesseur, le programme écrit en assembleur est transformé par une opération appelée ASSEMBLAGE en langage machine compréhensible par le microprocesseur.

La COMPILATION consiste à transformer une instruction compacte en langage évolué, par exemple WRITE, en une série d'instructions en langage machine, toujours les mêmes pour cette instruction, directement exécutables par le microprocesseur. Par exemple dans le cas de cette instruction WRITE, la transformation par COMPILATION aura pour objet de produire des instructions par lesquelles le microprocesseur devra, successivement, charger, dans un registre d'échange avec la mémoire, la valeur à écrire, sélectionner par son adresse une cellule de la mémoire où cette valeur doit être écrite, provoquer l'écriture, incrémenter son compteur d'instructions pour admettre une instruction suivante du programme, etc... Eventuellement le microprocesseur aura

dû aller lire la valeur écrite et la comparer à la valeur à écrire pour valider ou recommencer cette écriture dans le cas où le microprocesseur doit exécuter un protocole sécurisé d'écriture. On comprend bien qu'il est plus facile pour le programmeur d'écrire "WRITE" que d'écrire, en langage machine, toutes les instructions du microprocesseur.

Cependant, quand cela est nécessaire le programmeur écrit son programme dans un langage exécutable et compréhensible par le microprocesseur. Dans ce cas, plutôt que de lui imposer l'écriture fastidieuse des 1 et des 0 qui constituent, en langage machine, les seules expressions des instructions réellement exécutables par les microprocesseurs, on lui facilite la tâche en lui mettant à disposition un langage plus simple: le langage assembleur. Le langage assembleur est différent du langage évolué en ce sens qu'une instruction en langage assembleur est normalement transformée par l'opération d'ASSEMBLAGE en une seule instruction en langage machine, alors que la COMPILATION d'une instruction en langage évolué donne une série d'instructions en langage machine.

On pourra retrouver toutes ces notions relatives aux microprocesseurs dans le livre: "COMPRENDRE LES MICROPROCESSEURS" de Daniel QUEYSSAC, éditions RADIO, France 1983.

Dans le domaine des cartes à micro-circuit on est amené actuellement à demander aux programmeurs d'écrire leurs programmes en langage assembleur pour les raisons suivantes. Premièrement, pour ne pas occuper trop de place dans la mémoire programme de ces microprocesseurs, les programmes doivent être réduits à leur strict nécessaire. Ceci peut interdire l'emploi d'un langage évolué dont la traduction, au moment de la compilation,

peut conduire à un nombre d'instructions plus important que ce qui est réellement nécessaire. Par exemple, l'opération de vérification d'écriture évoquée ci-dessus pourra être systématiquement produite par la
5 COMPILATION, alors que, quand elle n'est pas justifiée, dans certains cas, on peut économiser de la place en mémoire en ne l'écrivant pas. Ceci conduit néanmoins à une difficulté supplémentaire de programmation puisqu'il faut faire attention à cette contrainte de place
10 limitée. Deuxièmement, la variété des microprocesseurs nécessiterait de devoir compiler les programmes avec des compilateurs variés, adaptés pour chaque langage à chaque type de microprocesseur. En pratique de tels compilateurs ne sont pas disponibles. Par ailleurs dans
15 ceux existants on constate une non allocation dynamique des variables des fonctions. Ceci implique, au vue des dimensions de la mémoire de travail une impossibilité d'utiliser la puissance des langages évolués (découpage en fonction par exemple).

20 La conséquence de cette situation est que les programmeurs de cartes à micro-circuit sont très rapidement intellectuellement liés au type de microprocesseur qu'ils connaissent bien. Donc il leur est peu facile de concevoir des nouvelles applications
25 quand le microprocesseur qu'ils connaissent n'est pas apte à les exécuter, par exemple parce que son jeu d'instructions n'a pas été prévu pour cela. Il leur est alors très difficile de changer leurs habitudes et de devenir aussi habiles et expérimentés avec un nouveau
30 microprocesseur qu'ils l'étaient avec un précédent microprocesseur. En outre, même la bonne connaissance du jeu d'instructions de plusieurs microprocesseurs ne peut pas donner à un programmeur une efficacité de travail qu'il aurait s'il écrivait ses programmes dans un

langage évolué, plus puissant et aussi connu par beaucoup d'autres programmeurs.

Ainsi si une application est écrite pour un microprocesseur donné, et si après cette écriture on décide d'utiliser un autre microprocesseur que celui pour lequel elle a été écrite et mise au point, on doit tout recommencer. Ceci est une perte de temps et d'argent.

Pour résoudre ces problèmes, dans l'invention, on a organisé le travail du microprocesseur d'une façon différente. Premièrement on utilise un langage évolué de programmation de type connu. Deuxièmement on utilise un compilateur de ce langage évolué de programmation pour produire, à partir d'un programme d'application écrit dans ce langage évolué, un programme en un langage intermédiaire. Ce langage intermédiaire sera un standard pour tous les microprocesseurs possibles. Cependant les instructions de ce programme intermédiaire ne sont pas exécutables par aucun des microprocesseurs, pas plus que les instructions du programme en langage évolué. On munit alors, dans chaque carte, le micro-circuit d'un programme interpréteur de commande. Un tel programme interpréteur de commande est susceptible de produire une série d'instructions écrites dans le langage du microprocesseur de ce micro-circuit et donc directement exécutables par ce microprocesseur, pour une instruction reçue dans ce langage intermédiaire. Ce programme interpréteur n'est ni un programme de COMPILATION ni un programme d'ASSEMBLAGE. En effet, l'interpréteur produit les instructions exécutables par le microprocesseur au fur et à mesure qu'il reçoit des instructions en langage intermédiaire d'une part, mais surtout il ne produit une autre série d'instructions exécutables par ce microprocesseur que lorsque les précédentes instructions

exécutables ont été exécutées. La production des instructions exécutables est donc faite en temps réel, au vol, au cours du déroulement du programme. Ces instructions directement exécutables n'existent donc
5 dans le déroulement du programme que d'une manière éphémère, que lorsqu'elles sont exécutées. Elles ne sont pas stockées comme telles dans une mémoire du micro-circuit.

La mise oeuvre de l'invention nécessite donc
10 l'écriture, une fois pour toutes, d'un programme de compilation intermédiaire pour compiler des instructions, écrites en un langage évolué, en des instructions en langage intermédiaire. Il y a cependant autant de programmes de compilation intermédiaires qu'il
15 y a de langages évolués. Actuellement le nombre des langages évolués couramment utilisés est de l'ordre d'une dizaine: c'est faible. Elle nécessite encore, mais là aussi une fois pour toute, l'écriture d'un programme interpréteur de commande pour le microprocesseur. Il y a
20 cependant également autant de programmes interpréteurs qu'il y a de microprocesseurs différents. On peut dénombrer actuellement une dizaine de microprocesseurs de sorte que seuls une dizaine de programmes interpréteurs doivent être écrits.

25 L'intérêt de l'invention est alors que n'importe quelle application, écrite en un langage évolué est alors exécutable sur n'importe quel microprocesseur. Sans l'invention c'est environ une dizaine de programmes exécutables qu'il aurait fallu écrire si on avait voulu
30 être sûr de couvrir toutes les possibilités (pour 10 microprocesseurs). Ceci aurait été très long puisque l'écriture et la mise au point d'un programme directement exécutable est longue. Par ailleurs la place occupée en mémoire est plus faible avec l'invention. A

titre d'exemple un programme de 1200 lignes écrit en langage C, compilé avec le compilateur C de BYTE CRAFT donne en langage exécutable par un microprocesseur un volume d'instructions égal à 8 kilooctets, ou 8 KO. Avec la compilation intermédiaire de l'invention, le programme intermédiaire correspondant occupe 4 KO. Sachant que le programme interpréteur occupe 2,1 KO en mémoire du micro-circuit, on a pu faire une économie de place d'environ 2 KO. Cette économie a été faite par ailleurs sans avoir à surveiller la suppression de partie d'instructions qui pouvaient, dans certains cas s'avérer inutiles. De plus, le programme essai qui sert de référence a d'abord été écrit en C-Byte (car plus contraignant d'un point de vue syntaxique, avec au maximum deux octets de déclaration). Ce langage C-Byte n'est pas particulièrement adapté aux systèmes portables. Ceci implique qu'en écrivant directement avec le compilateur de l'invention le gain serait encore supérieur.

L'invention a donc pour objet un support portable à micro-circuit dont le micro-circuit est muni d'un microprocesseur, d'une mémoire programme (ROM), d'une mémoire de données (EEPROM), et de moyens de faire exécuter par le microprocesseur un programme contenu dans la mémoire programme, caractérisé en ce que la mémoire programme comporte une zone dans laquelle est stocké un programme interpréteur correspondant au microprocesseur, pour faire exécuter par ce microprocesseur, une à une, les instructions d'un programme intermédiaire d'application chargé dans la mémoire programme ou la mémoire de données, après les avoir individuellement fait interpréter par le programme interpréteur, afin par exemple que ce programme intermédiaire d'application agisse sur des données

contenues dans la mémoire de données.

L'invention sera mieux comprise à la lecture de la description qui suit et à l'examen des figures qui l'accompagnent. Celles-ci ne sont données qu'à titre
5 indicatif et nullement limitatif de l'invention. Notamment, la référence faite à un langage de programmation particulier doit se comprendre comme transposable aux autres langages de programmation disponibles. De même la citation d'un microprocesseur
10 particulier ne peut être considérée comme une application de l'invention à ce seul microprocesseur.

A titre d'exemple on joint ici, en un langage compréhensible par le microprocesseur de type ST8 de SGS-THOMSON Microelectronics, un programme interpréteur
15 susceptible d'interpréter des instructions compilées en langage intermédiaire d'un programme d'application écrit dans un langage évolué par exemple C. Le compilateur de ce langage doit produire des instructions dont la liste est donnée à la dernière page de cette annexe. Cette
20 liste renseigne donc sur le niveau d'achèvement qui doit être conduit avec le compilateur. Pour en faciliter la compréhension, le programme *interpréteur* est écrit en assembleur. Il doit cependant être assemblé selon le programme d'assemblage du microprocesseur décrit avant
25 d'être exécutable.

Les figures montrent:

- Planche 1/8, figure 1 : un exemple de réalisation de l'invention;
- Planche 2/8, figure 2 : les étapes nécessaires pour
30 mettre en oeuvre le procédé de l'invention;
- Planche 3/8, figure 3 : une représentation détaillée d'un mode de fonctionnement de l'invention.
- Les planches 4/8 à 8/8 montrent, sous forme de court listage, des micro-instructions de l'interpréteur de l'invention.

La figure 1 montre un exemple de réalisation de l'invention. Un support portable 1, ici sous forme de carte à puce est prévu pour être inséré dans un lecteur 2 en relation avec une machine 3. Dans un exemple de type connu la machine 3 est un distributeur automatique de billets de banque. N'importe quelle autre application est cependant envisageable. Dans cet exemple la machine 3 est même munie d'un clavier 4 sur lequel peut intervenir un utilisateur, le titulaire de la carte 1, pour choisir de faire exécuter une option ou une autre d'un programme d'utilisation de la carte. Le programme d'utilisation de la carte a été introduit dans une partie 5 d'une mémoire morte 6. Normalement l'utilisateur de la carte n'a pas les moyens de modifier le programme contenu dans la mémoire 6. Ce programme y a été introduit par l'émetteur de la carte: la banque qui gère par ailleurs la machine 3. Ce programme a été écrit et mis au point par un programmeur de cette banque. Ce programme est destiné à permettre à l'utilisateur de la carte d'effectuer des opérations diverses: visualisation de compte, retrait d'argent liquide, passation d'ordre de virement, d'ordres d'achat en bourse ou autres.

La carte 1 comporte un micro-circuit électronique comportant un microprocesseur 7, la mémoire programme 6, une mémoire de données 8, et une mémoire de travail 9. La mémoire 9 est une mémoire à accès aléatoire de type statique ou dynamique. Elle est ici volatile. Le micro-circuit comporte encore un bus de données 10 et un bus d'adresses 11 pour permettre à ces mémoires et à ce microprocesseur d'échanger des informations, des données, entre eux et aussi avec un organe d'entrée-sortie 12. L'organe d'entrée-sortie 12 est capable de communiquer avec une interface correspondante dans le lecteur 2.

La mémoire de travail 9 est nécessaire dans l'invention, et le microprocesseur doit être conçu pour pouvoir exécuter des instructions qui lui sont présentées depuis cette mémoire. Certains microprocesseurs néanmoins sont aptes à exécuter des instructions (directement compréhensibles par ces microprocesseurs) et mémorisées telles quelles dans des mémoires mortes associées à ces microprocesseurs. Ceci n'est pas le cas dans l'invention où, comme on le verra plus loin, les instructions concernant le programme d'application ne sont pas mémorisées sous une forme directement exécutable par le microprocesseur (bien que pour des raisons de mémorisation elles soient elles aussi composées en code binaire de 1 et de 0).

Les instructions du programme d'application, dans l'invention, doivent être interprétées, au moyen d'un programme d'interprétation contenu (sous une forme directement exécutable par le microprocesseur 7) par exemple dans une autre partie 13 de la mémoire morte 6.

La mémoire 6 n'est pas nécessairement physiquement partagée en deux parties distinctes. Des adresses de différentes cellules de cette mémoire peuvent à elles seules permettre de distinguer le contenu de la mémoire 6: programme P représentant l'application, ou programme I représentant le programme interpréteur directement exécutable tel quel par le microprocesseur 7. De préférence, disposant du microprocesseur 7, on va faire interpréter par ce microprocesseur 7 les instructions du programme d'application. Il serait néanmoins possible de faire interpréter ces instructions par un autre microprocesseur, moins puissant mais physiquement voisin du microprocesseur 7 dans le micro-circuit de la carte 1. L'utilisation du même microprocesseur conduit à simplifier l'architecture du système comme on le verra

plus loin.

La figure 2 montre les étapes nécessaires pour mettre en oeuvre le procédé de l'invention. Premièrement, dans une phase 14 on écrit le programme de l'application comme si les problèmes cités relatifs aux cartes à micro-circuit étaient inexistants. On sait écrire de tels programmes: ils ne nécessitent que de connaître le fonctionnement de la machine 3 et les fonctions qu'on veut proposer aux utilisateurs. Dans un exemple de mise en oeuvre de l'invention le langage de programmation est pour l'essentiel le langage C, avec les options de programmation suivantes. En ce qui concerne la déclaration des variables, les types de données suivants sont autorisés: mode caractère "char" sur un octet, mode entier "int" sur deux octets, tableaux de caractères ou d'entiers à une dimension (avec un indice de 0 à n), et existence de pointeurs sur les caractères ou les entiers. En ce qui concerne les spécifications de classes de mémoires, on respecte celles déjà existantes, et on introduit les classes de stockages permettant d'accéder aux mémoires EPROM et/ou EEPROM. En ce qui concerne les expressions: toutes les expressions du langage C sont autorisées: notamment celles avec des opérateurs unaires et binaires, avec des opérateurs logiques et de décalage, et les expressions conditionnelles. En ce qui concerne les déclarations de fonction, elles sont toutes permises, avec passages de paramètres et allocation dynamique des variables locales. Toutes les structures de contrôle sont également acceptées: notamment les IF, WHILE, FOR, SWITCH etc.... Par ailleurs le langage C comporte déjà des fonctions externes facilitant l'interface système-matériel pour gérer les aspects matériels de la carte, notamment la gestion de son organe d'entrée-sortie 12.

Puis dans une étape 15 on compile le programme 14.

Le programme compilateur qui permet la production du programme d'application sous sa forme intermédiaire est un programme compilateur avec pour principales caractéristiques les caractéristiques suivantes. Un compilateur est un programme qui prend en entrée un fichier source de type texte et qui correspond au programme que l'on veut compiler, et qui produit en sortie un fichier en un langage différent. Dans l'état de la technique le langage différent est le langage machine (directement exécutable par le micro-processeur) ou éventuellement de l'assembleur. Dans l'invention, c'est un langage intermédiaire. Le fichier programme source est constitué de déclarations de variables, de pragmas, et de fonctions. Les variables sont repérées par leur identificateur, les pragmas permettent d'affecter à certaines variables des adresses en mémoire et de spécifier des aspects liés au processeur lui-même. Par exemple, la ROM ou l'EEPROM sont à telle ou telle adresse.

On veut, par exemple, compiler un fichier source où on va trouver une instruction.

Dans un premier temps on trouve dans ce fichier source la déclaration de la variable I. A cette variable I on attache ainsi un type de variable. Ce type est par exemple entier. Après la déclaration de variable on trouvera l'instruction

I = 14;

dans le programme ou dans une fonction. Après avoir vu la déclaration, cette séquence est correcte puisqu'on trouve un nom de variable I, un signe d'affectation =, une valeur ou une notation d'entier 14 et un point virgule ;. Le point virgule veut dire que l'instruction

est terminée. Cela veut dire que derrière le "14" il n'y a pas un "+1" ou un "+2". C'est donc la fin de l'instruction. Après avoir analysé l'instruction et sa syntaxe, on va avec le programme compilateur produire une expression en langage intermédiaire. Pour cette instruction $I = 14$, il faut empiler, mettre dans une pile, l'adresse de la variable I . On remarque qu'on connaît l'adresse de la variable I parce que quand on a exploré les déclarations, à chaque fois qu'il avait une nouvelle déclaration on allouait une adresse particulière à chaque variable. Après avoir empilé l'adresse de I , on empile au-dessus la valeur 14. Puis on va prendre la valeur qui est en haut de la pile (14) et l'affecter à l'adresse qui est contenue dans la pile juste en-dessous de cette valeur (14) du haut de la pile. Dans le cas présent, cette expression de langage intermédiaire comporte donc les trois instructions : empiler l'adresse de I , empiler la valeur (14) et mettre la valeur du sommet de la pile à l'adresse contenue dans le sous-sommet de la pile.

Pour le compilateur de l'invention, on s'est attaché à produire environ 70 instructions élémentaires (69 exactement) du type de chacune des trois précédentes. On produit donc 70 instructions qui doivent être interprétées. Ce nombre, de type empirique, et induits par les considérations suivantes. Si le compilateur fait beaucoup de chose, la complexité des instructions différentes à exécuter par l'interpréteur va diminuer pour atteindre le langage machine. La place occupée en mémoire par l'interpréteur va diminuer (un peu). Le bilan global sera moins intéressant. Dans le cas contraire c'est le programme interpréteur lui-même qui occupera une place prohibitive dans la mémoire. La puissance du langage intermédiaire est ainsi déterminée

empiriquement. C'est le fait d'expériences réussies et d'échecs. Si on a un langage intermédiaire qui est très puissant, on a automatiquement plus d'instructions dans le langage intermédiaire. Si on a plus d'instructions dans le langage intermédiaire on a automatiquement un interpréteur qui devient important. Puissant veut dire pour le compilateur d'être à même de gérer 3 ou 4 niveau de piles, les additionner, les multiplier. Il pourrait faire beaucoup plus de choses, On aurait alors un compilateur et un langage intermédiaire beaucoup plus puissant, mais il faudrait un interpréteur beaucoup plus long, parce qu'il y aurait beaucoup plus d'instructions qu'il faudrait traduire en langage du processeur réel. Donc le gain en place mémoire que l'on aurait obtenu en essayant de trouver un langage intermédiaire, on le perdrait parce qu'on aurait un interpréteur trop long. Un langage intermédiaire très puissant traiterait directement les instructions en langage évolué. Si l'interpréteur fait tout, il va être gros et occuper de la place en mémoire

Une fois que le programme intermédiaire est créé, on le stocke dans la partie 5 de la mémoire 6.

Puis, ou au préalable, on écrit en une étape 17 un programme interpréteur de commande, spécifique du microprocesseur utilisé dans la carte. Ce programme est pour le microprocesseur de l'exemple de l'invention celui montré dans l'annexe A. Il a les caractéristiques suivantes. C'est un programme qui est écrit dans le langage machine du microprocesseur (bien qu'il soit en annexe sous sa forme assembleur). En entrée, l'interpréteur prend les expressions produites par le compilateur et les transforme en des instructions directement compréhensible par le microprocesseur. Son algorithme principal est simple. On prend la première

expression trouvée, on l'exécute, et le pointeur de programme pointe sur l'expression suivante. Pour $I = 14$ on avait généré trois instructions : 1) empiler l'adresse de I ; 2) empiler la valeur 14 ; 3), affecter le sommet de pile à l'adresse contenue dans le sous-sommet de pile.

L'interpréteur prend ici la première instruction qui s'appelle "EMPILER l'adresse de I ". Empiler l'adresse de I consiste à mettre l'adresse de I sur une pile. L'instruction empilage de I est codée en langage machine sur trois octets. Le premier octet est le code opération, qui correspond à EMPILER. Le programme de l'interpréteur est tel que lorsqu'on trouve un "EMPILER", juste derrière, les deux octets qui suivent représentent l'adresse de l'endroit où on doit aller chercher ce qu'on doit empiler, et on sait donc qu'il faut mettre l'adresse sur la pile. Pour le microprocesseur décrit, cela donne une instruction appelée DJSR-EMPILER. Son code, qu'on retrouvera dans l'annexe A, est

`00000000 DJSR-EMPILER PUSH_BASE_DS...`

Il comporte trois instructions en langage machine : les instructions ici écrites en assembleur.

```

00000000 move b7,b4
00000001 jsr empiler
00000002 jmp décodage

```

Le programme interpréteur de commande, directement exécutable par le microprocesseur 7, est ensuite chargé en une étape 18 dans la partie 13 de la mémoire 6.

Pour le déroulement de l'application, le fonctionnement de l'invention est le suivant. Chaque instruction du programme intermédiaire (du programme stocké dans la partie 5 de la mémoire 6) est considéré comme une macro-instruction qui est décodée en une étape

19 à l'aide du programme interpréteur. Cette macro-instruction est équivalente à une suite de micro-instructions directement exécutables par le microprocesseur. Les micro-instructions de cette suite
5 sont exécutées à la suite les unes des autres jusqu'à la dernière de la suite. Dès que la dernière de la suite est exécutée, par une étape 21, le déroulement du programme retourne au décodage d'une macro-instruction suivante du programme intermédiaire.

10 La figure 3 permet de comprendre le fonctionnement de l'invention. Elle reprend les mêmes éléments que ceux déjà vus jusqu'ici. On a en plus fait figurer un compteur de programme 22 susceptible de permettre de traiter les unes après les autres les instructions IM du
15 programme intermédiaire contenu dans la mémoire 5. Une telle instruction de ce programme intermédiaire est par exemple codée sur trois octets: un octet contenant un code instruction et deux octets contenant une adresse d'un opérande. Dans un premier temps le code instruction
20 chargé en mémoire RAM 9 est envoyé à un décodeur d'instruction 23 du microprocesseur 7. L'interpréteur reconnaît le code instruction IM, en utilisant l'ALU 24 et le décodeur d'instructions 23, à cause de la présence de signaux de contrôle relatifs à ce stade de
25 l'exécution. Cette instruction est reconnue comme étant une instruction en langage intermédiaire. Dans ces conditions cette instruction décodée provoque le chargement dans la mémoire 9 de la série 27 des micro-instructions IP à IP + 4 dont la séquence a
30 correspondu au décodage de l'instruction IM. Ce chargement est provoqué par l'envoi de l'instruction IM au décodeur 23, à l'ALU 24 au sortir du décodeur 23, et dans une table d'adressage 241 au sortir de l'ALU 24. La table 241 peut aussi être remplacée par un petit sous

programme qui effectuerait le même travail. Le programme interpréteur comporte ainsi un certain nombre (69) de séries de micro-instructions, par exemple les séries 27 à 30. Au moyen d'un autre compteur d'instructions, on fait exécuter successivement les instructions IP à IP + 4. Ces instructions agissent sur l'opérande 25 contenu dans la mémoire de données 8 et dont l'adresse a été décodée par le registre d'adresse 26. Ces instructions IP à IP + 4 passent chacune à leur tour par le décodeur d'instruction 23 avant d'être envoyées à l'unité 24 où elles agissent sur l'opérande 25. On rappelle que les instructions IP à IP + 4 sont directement exécutables par l'unité 24. La fin d'une série de micro-instructions est marquée par la présence, dans chacune de ces séries d'instructions du programme interpréteur, d'une micro-instruction de fin ayant pour objet de provoquer des signaux de contrôle nécessaires pour passer à l'instruction en langage intermédiaire suivante. Cette pile de micro-instructions peut être facilement prise en compte par un microprocesseur 7 intégrant une gestion de pile. Elle peut aussi être simulée par les microprocesseurs n'ayant pas une telle gestion de pile directe. Cette fonction de pile est de préférence contenue dans le système d'exploitation du microprocesseur 7.

Pour la réalisation des algorithmes cryptographiques, en particulier dans le cas des applications bancaires, l'utilisation de l'assembleur pourra être préférable car elle permet d'obtenir des temps d'exécution courts. Ceci n'interdit pas d'utiliser l'invention, il suffit dans le déroulement du programme de faire appel à des sous-programmes écrits alors en langage machine (après ASSEMBLAGE). Ces sous programmes peuvent aussi être chargés dans la partie 13 de la mémoire 6.

REVENDICATIONS.

1 - Support portable à micro-circuit dont le micro-circuit est muni d'un microprocesseur, d'une mémoire programme (ROM), d'une mémoire de données (EEPROM), et de moyens de faire exécuter par le microprocesseur un programme contenu dans la mémoire programme, caractérisé en ce que la mémoire programme comporte une zone dans laquelle est stocké un programme interpréteur correspondant au microprocesseur, pour faire exécuter par ce microprocesseur, une à une, les instructions d'un programme intermédiaire d'application chargé dans la mémoire programme ou la mémoire de données, et après les avoir individuellement fait interpréter par le programme interpréteur, afin par exemple que ce programme intermédiaire d'application agisse sur des données contenues dans la mémoire de données.

2 - Support selon la revendication 1, caractérisé en ce que le programme interpréteur est un programme interpréteur susceptible de transformer un programme intermédiaire d'application compilé à partir d'un programme écrit dans un quelconque des langages de programmation suivant:

langage C
langage PASCAL
langage COBOL
langage BASIC
langage ADA
langage FORTRAN

3 - Support selon la revendication 1 ou la revendication 2, caractérisé en ce que la mémoire programme comporte en outre une zone contenant le

système d'exploitation du microprocesseur.

- 4 - Support selon l'une quelconque des revendications 1 à 3, caractérisé en ce que le micro-circuit comporte des moyens pour élaborer temporairement les instructions mises en oeuvre par le microprocesseur et une mémoire de travail pour les stocker pendant leur existence éphémère.

- 5 - Procédé d'utilisation d'un support à micro-circuit dont le micro-circuit est muni d'un microprocesseur, d'une mémoire programme (ROM), d'une mémoire de données (EEPROM), et de moyens (RAM) de faire exécuter par le microprocesseur un programme contenu dans la mémoire programme ou la mémoire de données, caractérisé en ce qu'il comporte les étapes suivantes:
- 15 - on charge un programme interpréteur de commandes dans la mémoire programme,
- on charge un programme intermédiaire d'application dans la mémoire de données,
- on fait interpréter au moins une instruction du programme intermédiaire par l'interpréteur de commande,
- 20 - on fait exécuter par le microprocesseur cette instruction intermédiaire interprétée,
- et on agit de même pour une instruction suivante du programme intermédiaire.

Pl.1/8

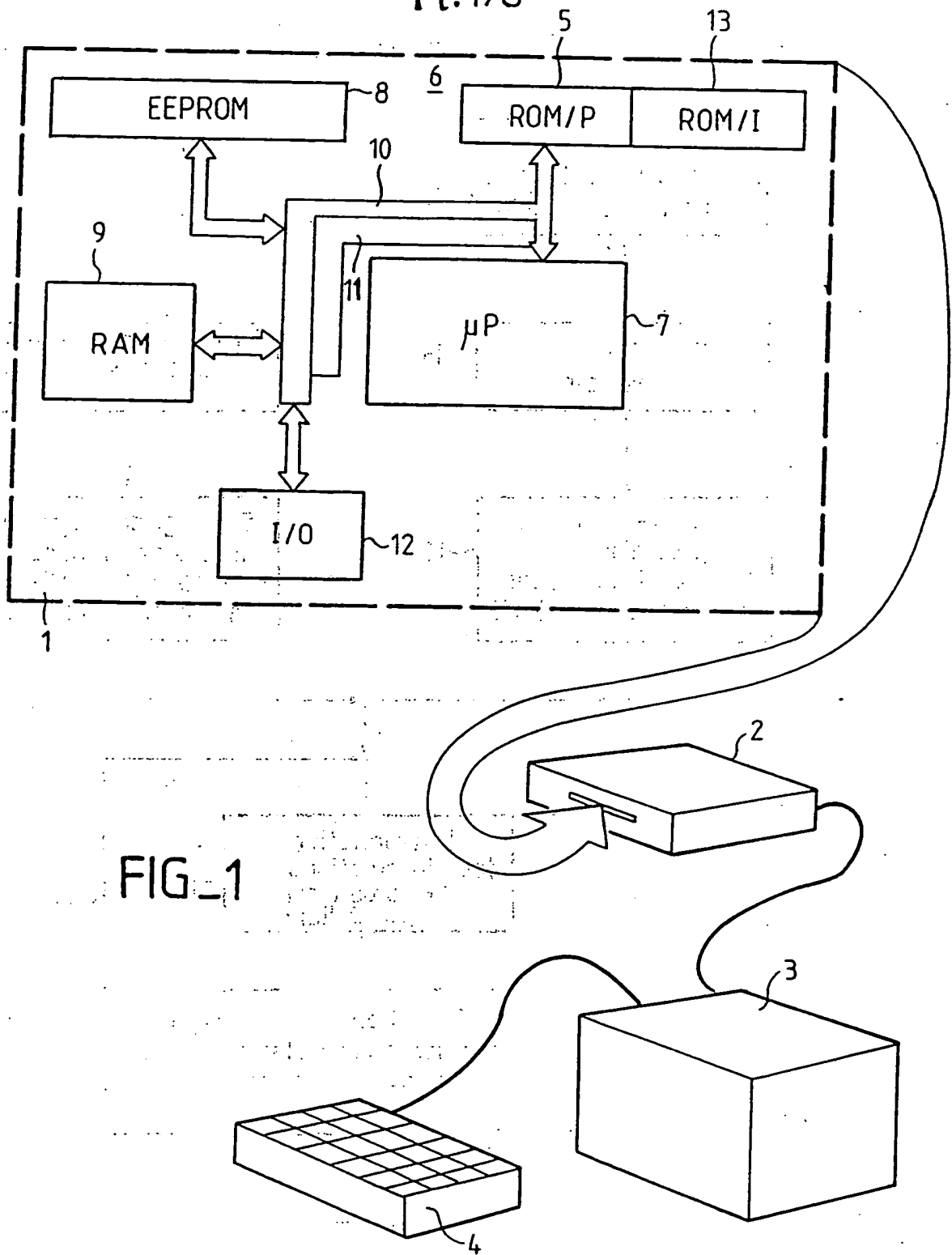
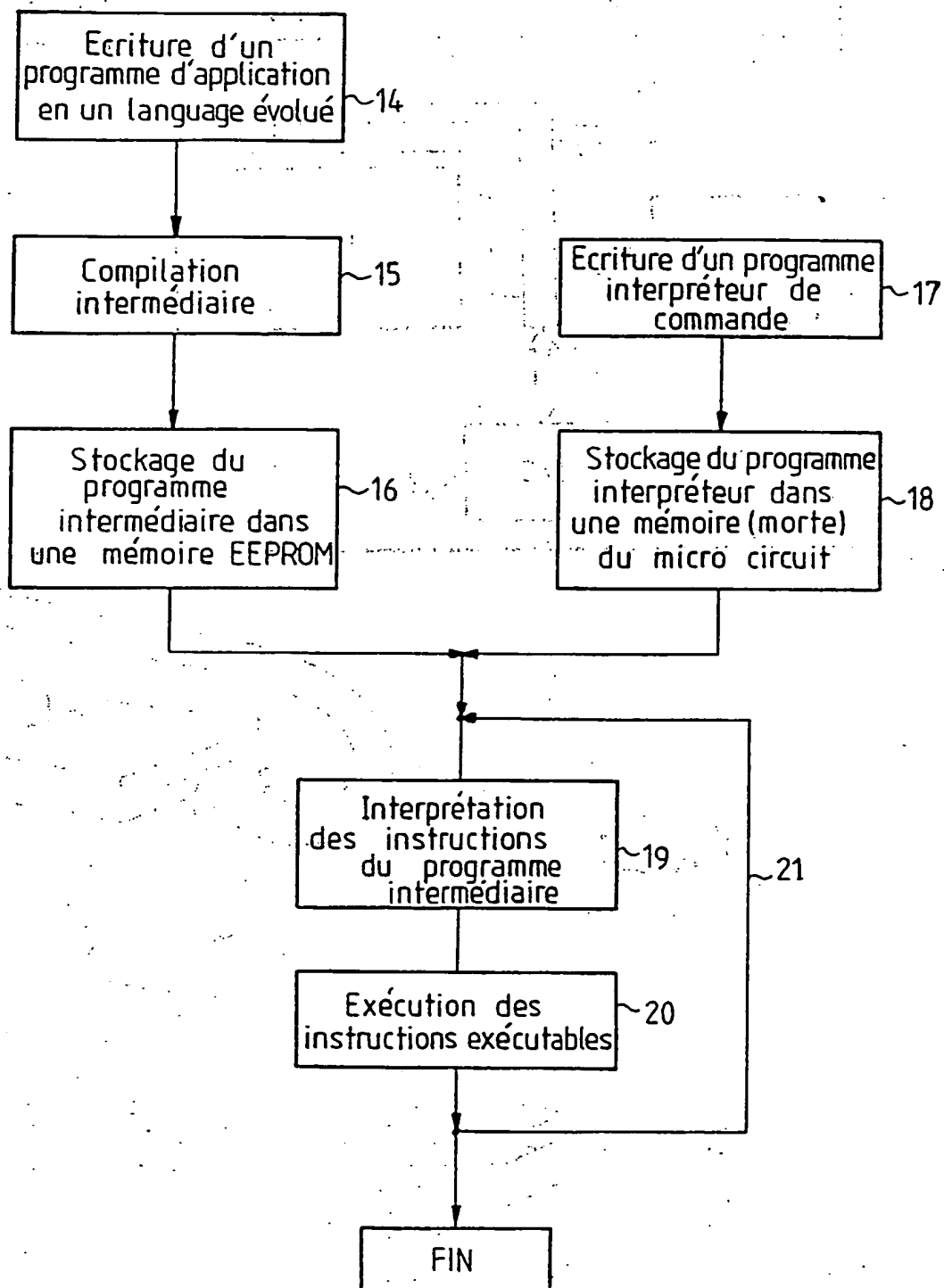


FIG. 1

Pl.2/8

FIG_2



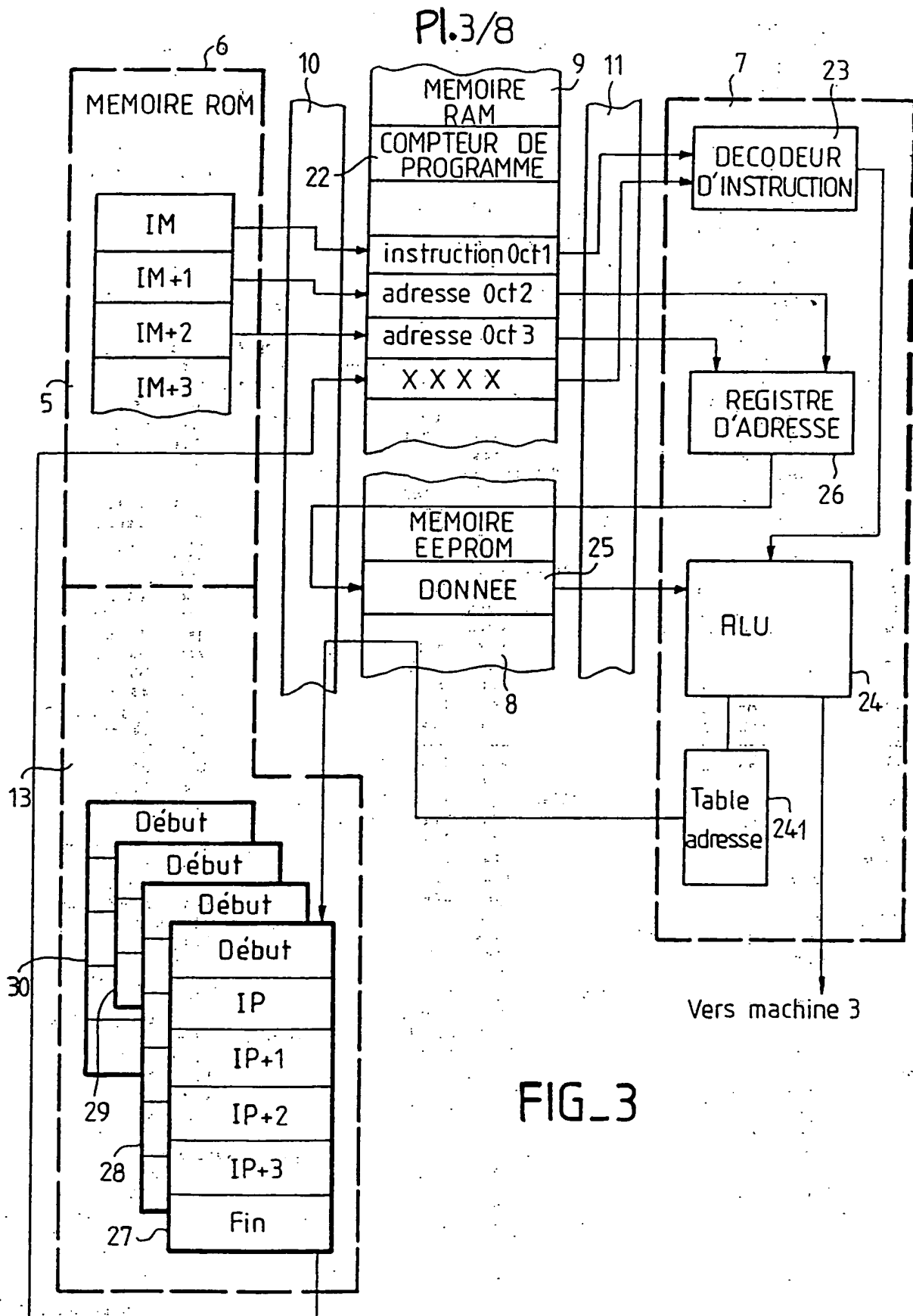


FIG. 3

INTERPRETEUR C_CARD, pour ASSEMBLEUR 6805

20 Janvier 1990

VERSION 1.0

```

a1 equ $40
a2 equ $100
a3 equ $09
a4 equ $00
move macro r,d
    lda d+1
    sta r+1
    lda d
    sta r
endm
addw macro r,d1,d2
    lda d1+1
    add d2+1
    sta r+1
    lda d1
    adc d2
    sta r
endm
subw macro r,d1,d2
    lda d1+1
    sub d2+1
    sta r+1
    lda d1
    sbc d2
    sta r
endm
.PAGE0
a5 rmb 2
b rmb 2
b1 rmb 2
b2 rmb 2
b3 rmb 1
b4 equ *
b5 rmb 1
b6 rmb 1
b7 rmb 2
b8 rmb 1
b9 rmb 1
cl rmb 2
rts rmb 1
.CODE
    org a2
    jmp ini
cbw equ *
    lda b5
    clr b5
    sta b6
    bpl cbw1
    dec b5
cbw1 equ *
    rts
store_word equ *
    lda $Sc7
    sta b9
    lda b7
    jsr b9
    inc cl+1
    bne store_word1
    inc cl
store_word1 equ *
    lda b7+1
    jsr b9
    rts
store_byte equ *
    lda $Sc7
    sta b9
    lda b5
    jsr b9
    rts
load_word equ *
    lda $Sc6
    sta b9
    jsr b9
    sta b7
    inc cl+1
    bne load_word1
    inc cl
load_word1 equ *
    jsr b9
    sta b7+1
    rts
load_byte equ *
    lda $Sc6
    sta b9
    jsr b9
    rts
empiler equ *
    ldx b+1
    decx
    decx
    stx b+1
    stx cl+1
    clr cl
    lda b7
    sta 0,x
    lda b7+1
    sta 1,x
    rts
depiler equ *
    ldx b+1
    lda 0,x
    sta b7
    lda 1,x
    sta b7+1
    incx
    incx
    stx b+1
    rts
depiltwo equ *
    jsr depiler
    move b4,b7
    jsr depiler
    rts
fetch equ *
    move cl,a5
    lda $Sc6
    sta b9
    jsr b9
    inc a5+1
    bne fetch1
    inc a5
fetch1 equ *
    rts
cmpw equ *
    lda b7+1
    sub b4+1
    sta b4+1
    bpl cmpw1
    lda b7
    sbc b4
    bmi cmpw2
    ora b4+1
    and $57F
    bra cmpw3
cmpw1 lda b7
    sbc b4
    cmpw2 ora b4+1
    cmpw3 rts
ini equ *
    lda $S81
    sta b9+3
    lda $a3
    sta a5
    lda $a4
    sta a5+1
    jsr fetch
    sta b3
    stop
    swi
    cmp $570
    blo aiguillage1
    jsr fetch
    sta b5
    lda b3
    cmp $580
    blo aiguillage2
    jsr fetch
    sta b6
    lda b3
    and $57F
    tax
    aslx
    lda tabcod3,x
    sta cl
    lda tabcod3+1,x
    bra aiguillagefinal
aiguillage1 equ *
    ldx b3
    aslx
    lda tabcod1,x
    sta cl
    lda tabcod1+1,x
    bra aiguillagefinal
aiguillagefinal equ *
    sta cl+1
    lda $Sc6
    sta b9
    jmp b9
aiguillage2 equ *
    lda b3
    and $58f
    tax
    aslx
    lda tabcod2,x
    sta cl
    lda tabcod2+1,x
    bra aiguillagefinal

```

Pl. 5/8

```

*
spush_immediat equ *
spush_base_ds equ *
    lda b5
    sta b7+1
    clr b7
    jsr empiler
    jmp decodage
debug equ *
    jsr depiler
    lda b7
    ldx b7+1
    swi
    jmp decodage
sval_word_bp equ *
    jsr cbw
    addw cl,b1,b4
    jsr load_word
    jsr empiler
    jmp decodage
spush_base_bp equ *
    jsr cbw
    addw b7,b1,b4
    jsr empiler
    jmp decodage
cmpvrai equ *
    lda #1
    bra fin_comparaison
cmpfaux equ *
    clra
fin_comparaison equ *
    sta b7+1
    clr b7
    jsr empiler
    jmp decodage
ega equ *
    jsr depiltwo
    jsr cmpw
    beq egavrai
    jmp cmpfaux
egavrai equ *
    jmp cmpvrai
dif equ *
    jsr depiltwo
    jsr cmpw
    bne difvrai
    jmp cmpfaux
difvrai equ *
    jmp cmpvrai
inf equ *
    jsr depiltwo
    jsr cmpw
    bmi infvrai
    jmp cmpfaux
infvrai equ *
    jmp cmpvrai
sup equ *
    jsr depiltwo
    jsr cmpw
    bhi supvrai
    jmp cmpfaux
supvrai equ *
    jmp cmpvrai
inf_ega equ *
    jsr depiltwo
    jsr cmpw
    bmi infegavrai
    beq infegavrai
    jmp cmpfaux
infegavrai equ *
    jmp cmpvrai

sup_ega equ *
    jsr depiltwo
    jsr cmpw
    bhi supegavrai
    beq supegavrai
    jmp cmpfaux
supegavrai equ *
    jmp cmpvrai
sadd_sp equ *
    lda b+1
    add b5
    sta b+1
    lda b
    add #0
    sta b
    jmp decodage
ret_fon equ *
    move b,b1
    jsr depiler
    move b1,b7
    jsr depiler
    move a5,b7
    jmp decodage
sval_byte_bp equ *
    jsr cbw
    addw cl,b1,b4
empiler_byte equ *
    jsr load_byte
    sta b7+1
    clr b7
    jsr empiler
    jmp decodage
val_byte equ *
    jsr depiler
val_bytel equ *
    move cl,b7
    jmp empiler_byte
val_byte_ds equ *
    move cl,b4
    jmp empiler_byte
val_byte_bp equ *
    addw cl,b1,b4
    jmp empiler_byte
sval_byte_ds equ *
    lda b5
    sta cl+1
    clr cl
    jmp empiler_byte
dup_stackb equ *
    jsr depiler
    jsr empiler
    jmp val_bytel
sto_byte equ *
    jsr depiler
    lda b7+1
    sta b5
    jsr depiler
    move cl,b7
    jsr store_byte
    jmp decodage
deb_fon equ *
    move b7,b1
    jsr empiler
    move b1,b
    jmp decodage

intrsys equ *
    move cl,b
    jsr load_word
    lda #6
    cmp b7+1
    bne code7
    lda b+1
    add #2
    sta cl+1
    lda b
    adc #0
    sta cl
    jsr load_word
    lda b7+1
    nop
    jmp decodage
code7 equ *
    lda #7
    cmp b7+1
    bne code8
    rti
    jmp decodage
code8 equ *
    lda #8
    cmp b7+1
    bne code9
    bil
    jmp decodage
code81 equ *
    jmp decodage
code9 equ *
    jmp decodage
val_word equ *
    jsr depiler
    move cl,b7
    jsr load_word
    jsr empiler
    jmp decodage
dup_stackw equ *
    jsr depiler
    jsr empiler
    move cl,b7
    jsr load_word
    jsr empiler
    jmp decodage
val_word_bp equ *
    addw cl,b1,b4
val_wordl equ *
    jsr load_word
    jsr empiler
    jmp decodage
val_word_ds equ *
    move cl,b4
    jmp val_wordl
sval_word_ds equ *
    lda b5
    sta cl+1
    clr cl
    jmp val_wordl
sto_word equ *
    jsr depiler
    move b4,b7
    jsr depiler
    move cl,b7
    move b7,b4
    jsr store_word
    jmp decodage
or_logical equ *
    jsr depiltwo
    tst b4+1
    beq or_logical1
    jmp cmpvrai

```

Pl. 6/8

```

or_logical1 equ *
    tst b7+1
    beq or_logical2
    jmp cmpvrai
or_logical2 equ *
    jmp cmpfaux
and_logical equ *
    jsr depiltwo
    tst b4+1
    beq and_logical1
    jmp cmpvrai
and_logical1 equ *
    jmp cmpfaux
not_logical equ *
    jsr depiler
    tst b7+1
    beq not_logical1
    jmp cmpfaux
not_logical1 equ *
    jmp cmpvrai
mul equ *
    jsr depiltwo
    move c1,b4
    ldx #16
    clr b4
    clr b4+1
    ror b7
    ror b7+1
    bcc mul2
    lda b4+1
    add c1+1
    sta b4+1
    lda b4
    adc c1
    sta b4
    mul2 ror b4
    ror b4+1
    ror b7
    ror b7+1
    decx
    bne mul1
    jsr empiler
    jmp decodage
mod equ *
    jsr depiltwo
    jsr div16
    jsr rmod
    jsr empiler
    jmp decodage
add equ *
indice_byte equ *
    jsr depiltwo
    addw b7,b7,b4
    jsr empiler
    jmp decodage
sub equ *
    jsr depiltwo
    subw b7,b7,b4
    jsr empiler
    jmp decodage
div equ *
    jsr depiltwo
    jsr div16
    jsr rdiv
    jsr empiler
    jmp decodage
shr equ *
    jsr depiltwo
    ldx b4+1
    shr1 lsr b7
    ror b7+1
    decx
    bne shr1
    jsr empiler
    jmp decodage
shl equ *
    jsr depiltwo
    ldx b4+1
    shl1 lsl b7+1
    rol b7
    decx
    bne shl1
    jsr empiler
    jmp decodage
and equ *
    jsr depiltwo
    lda b7+1
    and b4+1
    sta b7+1
    lda b7
    and b4
    sta b7
    jsr empiler
    jmp decodage
or equ *
    jsr depiltwo
    lda b7+1
    ora b4+1
    sta b7+1
    lda b7
    ora b4
    sta b7
    jsr empiler
    jmp decodage
xor equ *
    jsr depiltwo
    lda b7+1
    eor b4+1
    sta b7+1
    lda b7
    eor b4
    sta b7
    jsr empiler
    jmp decodage
neg equ *
    jsr depiler
    lda #0
    sub b7+1
    sta b7+1
    lda #0
    sbc b7
    sta b7
    jsr empiler
    jmp decodage
not equ *
    jsr depiler
    com b7+1
    com b7
    jsr empiler
    jmp decodage
inc_byte equ *
    jsr depiler
    move c1,b7
    jsr load_byte
    inca
    sta b5
    jsr store_byte
    jmp decodage
dec_byte equ *
    jsr depiler
    move c1,b7
    jsr load_byte
    deca
    sta b5
    jsr store_byte
    jmp decodage
inc_word equ *
    jsr depiler
    move c1,b7
    move b4,b7
    jsr load_word
    inc b7+1
    bne inc_word1
    inc b7
    inc_word1 equ *
    move c1,b4
    jsr store_word
    jmp decodage
dec_word equ *
    jsr depiler
    move c1,b7
    move b4,b7
    jsr load_word
    lda b7+1
    sub #501
    sta b7+1
    lda b7
    sbc #00
    sta b7
    dec_word1 equ *
    move c1,b4
    jsr store_word
    jmp decodage
indice_word equ *
    jsr depiler
    asl b7+1
    rol b7
    move b4,b7
    jsr depiler
    addw b7,b7,b4
    jsr empiler
    jmp decodage
push_ax equ *
    move b7,b2
    jsr empiler
    jmp decodage
pop_ax equ *
    jsr depiler
    move b2,b7
    jmp decodage
deb_fon_alloc equ *
    move b7,b1
    jsr empiler
    move b1,b
    subw b,b,b4
    jmp decodage
sdeb_fon_alloc equ *
    move b7,b1
    jsr empiler
    move b1,b
    lda b+1
    sub b5
    sta b+1
    lda b
    sbc #0
    sta b
    jmp decodage

```

Pl. 7/8

```

push_base_bp equ *
    addw b7,b1,b4
    jsr empiler
    jmp decodage
push_base_cs equ *
    addw b7,a5,b4
    jsr empiler
    jmp decodage
spush_base_cs equ *
    lda a5+1
    add b5
    sta b7+1
    lda a5
    adc #0
    sta b7
    jsr empiler
    jmp decodage
push_immediat equ *
push_base_ds equ *
    move b7,b4
    jsr empiler
    jmp decodage
dup_stack equ *
    jsr depiler
    jsr empiler
    jsr empiler
    jmp decodage
debut equ *
    clr b+1
    clr b
    clr b1
    clr b1+1
    jmp decodage
add_sp equ *
    addw b,b,b4
    jmp decodage
jmp equ *
jmp1 equ *
    addw a5,a5,b4
    jmp decodage
sjmp equ *
    jsr cbw
    jmp jmp1
jcf equ *
    jsr depiler
    tst b7+1
    beq jmp1
    jmp decodage
sjcf equ *
    jsr depiler
    tst b7+1
    beq sjmp
    jmp decodage
jcv equ *
    jsr depiler
    tst b7+1
    bne jmp1
    jmp decodage
sjcv equ *
    jsr depiler
    tst b7+1
    bne sjmp
    jmp decodage
call equ *
    move b7,a5
    jsr empiler
    addw a5,a5,b4
    jmp decodage

scall equ *
    move b7,a5
    jsr empiler
    jsr cbw
    addw a5,a5,b4
    jmp decodage
fin equ *
    wait
div16 equ *
    clr b8
    clr cl
    clr cl+1
    incx
div161 equ *
    lsl b7+1
    rol b7
    rol cl
    rol cl+1
    lda cl
    sub b4+1
    sta cl
    lda cl+1
    sbc b4
    sta cl+1
    bcc div162
    lda b4+1
    add cl
    sta cl
    lda b4
    adc cl+1
    sta cl+1
    sec
div162 equ *
    rolx
    rol b8
    bcc div161
    rts
rdiv equ *
    comx
    stx b7+1
    ldx b8
    comx
    stx b7
    rts
rmod equ *
    ldx cl+1
    stx b7
    lda cl
    sta b7+1
    rts

tabcod1 equ *
    dw 0
    dw debut 1
    dw fin 2
    dw deb_fon 3
    dw ret_fon 4
    dw sto_byte 5
    dw sto_word 6
    dw inc_byte 7
    dw inc_word 8
    dw dec_byte 9
    dw dec_word 0x0a
    dw val_byte 0x0b
    dw indice_byte 0x0c
    dw indice_word 0x0d
    dw and_logical 0x0e
    dw or_logical 0x0f
    dw or 0x10
    dw xor 0x11
    dw and 0x12
    dw ega 0x13
    dw dif 0x14
    dw inf 0x15
    dw sup 0x16
    dw inf_ega 0x17
    dw sup_ega 0x18
    dw shr 0x19
    dw shl 0x1A
    dw add 0x1B
    dw sub 0x1C
    dw mul 0x1D
    dw div 0x1E
    dw mod 0x1F
    dw neg 0x20
    dw not_logical 0x21
    dw not 0x22
    dw val_word 0x23
    dw push_ax 0x24
    dw pop_ax 0x25
    dw dup_stack 0x26
    dw dup_stackb 0x27
    dw dup_stackw 0x28
    dw intrsys 0x29
    dw debug 0x2A
tabcod2 equ *
    dw sdeb_fon_alloc 0x70
    dw spush_immediat 0x71
    dw spush_base_bp 0x72

```

Pl. 8/8

```

tabcod1 equ *
dw 0
dw debut 1
dw fin 2
dw deb_fon 3
dw ret_fon 4
dw sto_byte 5
dw sto_word 6
dw inc_byte 7
dw inc_word 8
dw dec_byte 9
dw dec_word 0x0a
dw val_byte 0x0b
dw indice_byte 0x0c
dw indice_word 0x0d
dw and_logical 0x0e
dw or_logical 0x0f
dw or 0x10
dw xor 0x11
dw and 0x12
dw ega 0x13
dw dif 0x14
dw inf 0x15
dw sup 0x16
dw inf_ega 0x17
dw sup_ega 0x18
dw shr 0x19
dw shl 0x1a
dw add 0x1b
dw sub 0x1c
dw mul 0x1d
dw div 0x1e
dw mod 0x1f
dw neg 0x20
dw not_logical 0x21
dw not 0x22
dw val_word 0x23
dw push_ax 0x24
dw pop_ax 0x25
dw dup_stack 0x26
dw dup_stackb 0x27
dw dup_stackw 0x28
dw intrsys 0x29
dw debug 0x2a

```

```

*
tabcod2 equ *
dw sdeb_fon_alloc 0x70
dw spush_immediat 0x71
dw spush_base_bp 0x72
dw sjmp 0x73
dw scall 0x74
dw sadd_sp 0x75
dw spush_base_cs 0x76
dw spush_base_ds 0x77
dw sjcf 0x78
dw sval_word_bp 0x79
dw sval_word_ds 0x7a
dw sval_byte_bp 0x7b
dw sval_byte_ds 0x7c
dw sjcv 0x7d

```

```

*
tabcod3 equ *
dw deb_fon_alloc 0x80
dw push_immediat 0x81
dw push_base_bp 0x82
dw jmp 0x83
dw call 0x84
dw add_sp 0x85
dw push_base_cs 0x86
dw push_base_ds 0x87
dw jcf 0x88
dw val_word_bp 0x89
dw val_word_ds 0x8a
dw val_byte_bp 0x8b
dw val_byte_ds 0x8c
dw jcv 0x8d
end

```

INSTITUT NATIONAL
de la
PROPRIETE INDUSTRIELLE

RAPPORT DE RECHERCHE
établi sur la base des dernières revendications
déposées avant le commencement de la recherche

FR 9011818
FA 451262

DOCUMENTS CONSIDERES COMME PERTINENTS		Revendications concernées de la demande examinée
Catégorie	Citation du document avec indication, en cas de besoin, des parties pertinentes	
Y	COMMUNICATIONS OF THE ACM vol. 26, no. 9, septembre 1983, pages 654-660; A.S. TANNENBAUM et al.: "A practical tool kit for making portable compilers" * page 654, colonne de gauche, ligne 1 - page 655, colonne de gauche, ligne 13; page 660, colonne de gauche, lignes 1-13 *	1-5
Y	DE-A-3 518 139 (SHARP K.K.) * en entier *	1-5
A	MICROPROCESSING & MICROPROGRAMMING vol. 21, nos. 1-5, août 1987, pages 275-282, Amsterdam, NL; K. WADA et al.: "Intermediate code for the sequential prolog machine PEK" * page 275, colonne de gauche; page 278, colonne de droite *	1-5
A	EP-A-0 331 754 (FANUC LTD.) * en entier *	1-5
A	DE-A-3 145 080 (SCHWARZ) * en entier *	1-5
A	US-A-4 443 865 (SCHULTZ et al.) * en entier *	1-5
A	US-A-4 618 925 (BRATT et al.) * en entier *	1-5
A	US-A-4 823 257 (TONOMURA) * en entier *	1-5
A	DD-A- 236 192 (FRICKE) * en entier *	1-5
Date d'achèvement de la recherche		Examineur
04-06-1991		DURAND J.
<p>CATEGORIE DES DOCUMENTS CITES</p> <p>X : particulièrement pertinent à lui seul Y : particulièrement pertinent en combinaison avec un autre document de la même catégorie A : pertinent à l'encontre d'au moins une revendication ou arrière-plan technologique général O : divulgation non-écrite P : document intercalaire</p> <p>T : théorie ou principe à la base de l'invention E : document de brevet bénéficiant d'une date antérieure à la date de dépôt et qui n'a été publié qu'à cette date de dépôt ou qu'à une date postérieure. D : cité dans la demande L : cité pour d'autres raisons</p> <p>& : membre de la même famille, document correspondant</p>		

REPUBLIC OF FRANCE

NATIONAL INSTITUTE
OF INDUSTRIAL PROPERTY
PARIS

Publication No.: 2 667 171
(To be used only for ordering copies)

National Registration No.: 90 11818

Int Cl³: G 06 F 9/06, 15/21; G 06 K 19/07

PATENT APPLICATION FOR A NEW INVENTION

Date Filed: 9-25-90

Applicant(s): Company known as GEMPLUS CARD
INTERNATIONAL, Limited Liability Company - FR.

Priority:

Inventor(s): Gordons Edouard, Grimonprez Georges, and
Paradinas Pierre

**Date of Public Disclosure
of the Application:** 03-27-92, Bulletin 92/13

**List of Documents cited in
the Literature Search Report:** *See end of this document.*

References to other related national documents:

Holder(s):

Representative: Law Firm of Ballot-Schmit

Portable support medium for an easily programmable microcircuit and programming procedure for said microcircuit.

To solve the programming problems associated with smart cards, the microcircuit in these cards is equipped with a command-interpretation program. Furthermore, application programs are compiled in an intermediate language that can be understood by the command interpreter. It is shown that in addition to saving memory space, this approach facilitates the programmer's task which will now be reduced to coding applications in higher-level languages which, a priori, are not suited for programming microprocessor cards.

PORTABLE SUPPORT MEDIUM FOR AN EASILY PROGRAMMABLE MICROCIRCUIT AND PROGRAMMING PROCEDURE FOR SAID MICROCIRCUIT

This invention was developed in collaboration with the LILLE COMPUTER SCIENCE LAB and the CENTER FOR MEDICAL COMPUTING STUDIES AND RESEARCH, which are part of the Université des Sciences et Techniques de Lille (Science and Technology University of Lille) and the Université de Droit et de la Santé de Lille (Law and Health Sciences University of Lille), respectively. The purpose of this invention has been to develop a microcircuit support medium, in which the microcircuit can be easily programmed, and a procedure for programming the microcircuit. More specifically, this invention has applications in the area of smart cards. In this case, the support medium is a card having the dimensions of a credit card. The purpose of the invention is to make the working capabilities provided by such support media available to programmers who are faced with developing various types of applications which must have a portable character. By portable character, we mean that the card or, more generally, any support medium of small dimensions (a few centimeters) and light weight (a few hundreds of grams), can be inserted in a card reader and perform a transaction between a machine connected to the reader and the card. This transaction is performed according to a certain protocol and instructions contained in the card.

The main difficulty associated with microcircuit cards comes from the fact that the microprocessor with which they are equipped has a small-capacity working storage (static or dynamic RAM), sometimes as little as 128 bytes, and a very small-capacity program memory (ROM: most of the time of the type EPROM or EEPROM), generally limited to a few tens of kilobytes and sometimes as little as a few kilobytes. Furthermore, card microprocessors have generally a limited set of instructions. These microprocessors can even be assimilated sometimes to micro-controllers whose data exchange buses are not all accessible from the micro-circuit external environment. The variety of possible designs for the main processing unit of these smart card microprocessors has led to the appearance of a large number of different microprocessors on the market. We note that such is not the case for large microprocessors (containing a large set of instructions) whose complexity has led only to a very limited number of designs due to the limited number of companies able to manufacture them.

In the case of microcircuit cards, the program corresponding to the application being implemented is usually stored in a non-volatile read-only memory built into the micro-circuit.

This wide variety of available microprocessors requires the programmer who wishes to use a microcircuit card to be perfectly familiar with the machine language of the microprocessor he/she is planning to use. This is not possible when one would like to be able to use several different types of microprocessors. In addition, due to the limited number of the instructions that can be executed by microprocessors in smart cards, the programming languages known as higher-level languages may not be entirely usable and should be tested. Among these higher-level languages, we shall later refer, for illustration purposes, to the languages known as C, COBOL, PASCAL, BASIC, ADA, as well as to numerous other languages which are equally known.

We remind the reader that the programming task of a human programmer is made easier by the use of these higher-level languages. These languages are both closer to the spoken language (instructions are clear: for instance, WRITE, IF, GO TO ...) and powerful because each instruction in these languages is meant to be a short-cut (which cannot be executed as is by the card's microprocessor) and can later be automatically converted in a series of machine-language

instructions that can be understood and executed by the microprocessor. This conversion is performed after the programming stage through an operation called **COMPILATION**. When the programming language used is not a higher-level language but a mnemonic language known as assembly language and close to that of the microprocessor, the program is converted through a process called **ASSEMBLY** of the program in machine language that can be understood by the microprocessor.

COMPILATION consists of converting a compact instruction written in a higher-level language, for instance **WRITE**, into a series of instructions in machine language, which are always the same for the given instruction and which can be directly executed by the microprocessor. For instance, in the case of the **WRITE** instruction, the **COMPILATION** will consist of producing instructions that tell the microprocessor to successively load the value to be written in an exchange register, select by its address the memory cell where this value is to be written, perform the writing operation, increment its instruction counter to receive the next program instruction, etc. Also, in cases where the microprocessor has to execute a checking protocol, it has to read the written value, compare it with the value to be written, and validate or redo the writing procedure. It is therefore clear that it would be easier for the programmer to type "**WRITE**" rather than the corresponding machine-language instructions for the microprocessor.

However, when necessary, the programmer has to write his program in a language that can be understood and executed by the microprocessor. In such cases, instead of requiring the programmer to go through the tedious exercise of typing 1's and 0's, which are the only forms of instructions that can actually be executed by the microprocessor, his/her programming task is made easier by the availability of a simpler language known as the assembly language. This language is different from a higher-level language in so far as each instruction of assembly language is converted through the **ASSEMBLY** process into a single instruction of machine language, whereas the **COMPILATION** of a higher-language instruction gives a series of machine language instructions.

All this information on microprocessors can be found in the book entitled: "**COMPRENDRE LES MICROPROCESSEURS**" (Understanding Microprocessors) by Daniel QUEYSSAC, RADIO editions, France, 1983.

In the area of microcircuit cards, programmers are currently asked to write their programs in assembly language for the following reasons. First, the program size should be reduced to the bare minimum, to avoid occupying too much space in the microprocessor's program memory. This may preclude the use of a higher-level language whose translation, during the compilation process, can lead to a number of instructions that is larger than the minimum number required. For instance, the check performed on writing operations, which was mentioned above, can be systematically generated by the **COMPILATION** process, whereas in some cases in which such a check is not justified, one can save memory space by not executing it. However, paying attention to this limited-space constraint adds to the difficulty of the programming task. Second, the existence of a variety of microprocessor types requires, for each language, the use of different compilers which are adapted to the specific type of microprocessor used. In practice, such compilers are not available. Moreover, in those compilers that are available, there is no dynamic allocation of function variables. Given the dimensions of the working storage, this implies that it is impossible to fully take advantage of the power of higher-level languages (such as subdividing the program into different functions).

This situation has led smart card programmers to become quickly attached, from an intellectual standpoint, to the type of microprocessor with which they are most familiar. Therefore, they find it difficult to design new applications when the microprocessor with which they are

familiar cannot execute the corresponding programs because, for example, its set of instructions has not been designed to do so. At that point, it is very difficult for them to change their habits and become as skilled and experienced with a new type of microprocessors as they were with the previous one. Furthermore, even a good knowledge of the different sets of instructions accepted by various types of microprocessors cannot provide the programmer with the same level of working efficiency as that which he/she would have when writing their programs in a higher-level language that is more powerful and equally known by many other programmers.

Hence, if an application is written for a given microprocessor and it is then decided to switch to another microprocessor, the whole process has to be repeated from the beginning. It is a waste of time and money.

To solve these problems, the work of the microprocessor has been structured differently in this invention. First, a known higher-level programming language is used. A compiler of this higher-level language is then used to convert the application program into a program written in an intermediate language. The latter shall be standard for all types of microprocessors. However, the instructions of this intermediate program cannot be directly executed by any microprocessor just as it is the case for the instructions of the program in the higher-level language. In each card, the microcircuit is then equipped with a command-interpretation program. For each instruction received in the intermediate language, the command-interpretation program can produce a series of instructions written in the language of the microprocessor used in the microcircuit and, therefore, directly executable by the microprocessor. This interpreting program is neither a **COMPILATION** program nor an **ASSEMBLY** program. First of all, the interpreter generates the microprocessor executable instructions as it receives the instructions in the intermediate language, and most of all, it generates a new set of executable instructions only after the preceding set of instructions has been executed. Therefore, executable instructions are generated in real time, on-the-fly, as the program is being run. When the program is being run, these directly executable instructions exist only for a short period of time, when they are being executed. They are not stored as such in the memory of the microcircuit.

Therefore, the implementation of the invention requires writing, once and for all, an intermediate compilation program to convert the instructions of the higher-level language into instructions written in the intermediate language. However, there will be as many intermediate compilation programs as there are higher-level languages. At present, the number of commonly used higher-level languages is around ten, which is a small number. The implementation of the invention also requires writing, once and for all, a command-interpretation program for the microprocessor. Again here, there will be a different interpreting program for each type of microprocessor. There are currently about ten different types of microprocessors so that only ten interpreting programs need to be written.

The point of the invention is that any application written in a higher-level language can now be executed on any microprocessor. Without the invention, it would have been necessary to write ten executable programs in order to cover all possible cases (i.e. for the 10 types of microprocessors). This would have been a very lengthy task, since writing and finalizing a directly executable program takes a long time. Furthermore, the memory space required for the program is smaller with the invention. For instance, in the case of a 1200 line program written in C and compiled using the C compiler of **BYTE CRAFT**, the volume of directly executable instructions is equal to 8 kilobytes or 8 kB. With the intermediate compilation process of this invention, the corresponding intermediate program occupies only 4 kB. Since the interpreting program occupies 2.1 kB of the micro-circuit

memory, the overall memory savings amount to about 2 kB. We note that these savings were achieved without having to monitor the suppression of those instructions which may not be needed all the time. Moreover, the test program which is used as a reference was first written in C-BYTE (which imposes more syntax restrictions, with a maximum of 2 bytes allowed for declarations). This C-BYTE language is not particularly suited for use in portable systems. This means that if the program is written directly with the invention's compiler, the memory savings would be even greater.

This invention is therefore concerned with the development of a microcircuit support medium in which the microcircuit contains a microprocessor, a read-only program storage (ROM), an electrically-erasable programmable read-only data storage (EEPROM), and means to allow the microprocessor to execute a program contained in the program storage; this support medium is also characterized by the fact that the program storage contains a section where a command-interpretation program, corresponding to the type of microprocessor used, is stored; the function of this command-interpretation program is to make the microprocessor execute, one by one, the instructions of an intermediate application program loaded either in the program storage or in the data storage, after these instructions have been individually interpreted by the command-interpretation program, for instance to make the intermediate application program act on data stored in the data storage.

The invention will be better understood after reading the following description and examining the accompanying figures. These figures are given only for illustrative purposes and shall in no way be interpreted in a restrictive manner. For instance, references made to a specific programming language should be understood as being transposable to other available programming languages. Similarly, reference to a specific type of microprocessors should not be understood as describing an application of this invention to only this type of microprocessors.

As an example, we are attaching to this document a program designed for interpretation into a language that can be understood by the ST8 microprocessor of SGS-THOMSON Microelectronics and capable of interpreting instructions compiled into an intermediate language and corresponding to an application program written in a higher-level language such as C. The compiler of the higher-level language should generate the instructions listed on the last page of the attachment. This list gives an idea of the level of completion to be achieved with the compiler. To make the command-interpretation program easier to understand, it was written in assembly language. Therefore, it has to be assembled using the microprocessor's assembly program before it can be executed.

The figures show:

- Plate 1/8, Figure 1: A Sample Implementation of the Invention;
- Plate 2/8, Figure 2: Necessary Steps for Implementing the Invention's Process;
- Plate 3/8, Figure 3: Detailed Representation of One of the Invention's Operation Modes.
- Plates 4/8 through 8/8 show, in the form of short listings, micro-instructions of the invention's interpreter.

Figure 1 shows a sample implementation of the invention. A portable support medium 1, illustrated here in the form of a smart card, is designed for insertion into a card reader 2 which is connected to a machine 3. In one example that is widely known, machine 3 is an automatic telling machine. However, all kinds of different applications are also possible. In our example, machine 3 is even equipped with a keyboard 4 which can be used by the user, the holder of card 1, to select the execution of any option available in the card's program. The card's program has been loaded in an area 5 of a read-only memory 6. Normally, the card's user cannot modify the program stored in memory 6. This program has been stored in memory by the card issuer: the bank that also operates

machine 3. The program has been written and finalized by a programmer working for that bank. The program is designed to allow the card's user to perform various operations: visualization of account status, cash withdrawals, performing transfer orders, stock purchasing orders, etc.

Card 1 has an electronic microcircuit containing a microprocessor 7, a program storage 6, a data storage 8, and a working storage 9. Storage 9 is a static or dynamic random access memory. In our example, it is volatile. The microcircuit also contains a data bus 10 and an address bus 11 which are used for information and data exchanges between the storages and the microprocessor, and between these components and an input/output unit 12. This unit is capable of communicating with a corresponding interface in the card reader 2.

Working storage 9 is necessary in this invention and the microprocessor should be designed to be able to execute instructions which are sent to it from this memory. Some microprocessors are able to execute instructions which they can directly understand and which are stored as such in read-only memory sections associated with the microprocessor. That is not the case in this invention where, as it shall be shown later, the instructions of the application program are not stored in a form that is directly executable by the microprocessor (even though, for storage purposes, they are set up in a binary code of 1's and 0's).

In the invention, the instructions of the application program must be interpreted by an interpretation program stored (in a form that is directly executable by microprocessor 7) for instance in another section 13 of the read-only memory 6. It is not necessary that memory 6 be physically divided into two different parts. Using only the addresses of different memory cells, it is possible to distinguish between the contents of these cells in memory 6: program P representing the application or program I representing the interpretation program which is directly executable by microprocessor 7. Having microprocessor 7 available, it is preferable to interpret the instructions of the application program using this microprocessor. It would be however possible to have these instructions interpreted by another microprocessor, less powerful but physically located close to microprocessor 7 in the microcircuit of card 1. The use of a single microprocessor results in a simpler system architecture as we shall see below.

Figure 2 shows the steps necessary for implementing the invention's process. First, in Step 14, the application program is written as if the problems mentioned in connection with microcircuit cards did not exist. People know how to write such programs: one needs only to know the mode of operation of machine 3 and the functions that will be made available to users. In one possible implementation of the invention, the programming language is essentially the C language, with the following programming options. Regarding the declaration of variables, the following data types are allowed: type "char" on one byte, type integer "int" on two bytes, one dimensional arrays of characters or integers (with an index running from 0 to n), and existence of pointers to characters and integers. Regarding memory class specifications, all existing ones are kept, and additional storage classes are introduced to access EPROM and/or EEPROM sections. Regarding expressions: all C expressions are allowed: in particular those containing unary and binary operators, logical and shift operators, and conditional expressions. All function declarations are allowed, with parameter passing and dynamic allocation of local variables. All control statements are also allowed: in particular, IF, WHILE, FOR, SWITCH, etc. In addition, the C language already contains external functions for easy software-hardware interfacing, which can be used to manage the hardware aspects of the card, in particular its input/output unit 12.

In Step 15, program 14 is compiled. The compilation program which generates the intermediate version of the application program has the following main characteristics. A compiler

is a program which takes as input a textual source file corresponding to the program to be compiled and which generates as output a file in a different language. As things stand now, the language of the output file is machine language (directly executable by the microprocessor) or assembly language. In the invention, it is an intermediate language. The file containing the source code consists of variable declarations, pragmas, and functions. Variables are identified by their names and pragmas are used to assign memory addresses to certain variables and to specify other aspects related to the processor itself. For instance, the ROM or EEPROM sections are located at such or such address.

Say for instance, that we wish to compile a source file containing the following instruction:

I = 14.

First, the declaration of variable I is found in the source code. A variable type is thus assigned to I. This type is for instance integer. After the variable declaration, the instruction I = 14; is found in the program or in a function. After the variable declaration has been detected, this syntax sequence is correct since it contains a variable name I, an assignment operator =, an integer value or notation 14 and a semi-colon ;. The semi-colon indicates the end of the instruction. This means that after the "14" there is no "+1" or "+2". It therefore marks the end of the expression. After having analyzed the instruction and its syntax, the compiler will generate an expression in the intermediate language. For this instruction, I = 14, the address of variable I has to be put on a stack. We note that the address of variable I is known because when declarations were being explored, a specific address was allocated to each variable. After having placed the address of I on a stack, the value 14 is stacked on top of it. The value located at the top of the stack (14) is then taken and assigned to the address located just underneath it in the stack. In this case, there will be three instructions in the intermediate language: place the address of I on the stack, place the value (14) on top of it, and take the value located at the top of the stack to the address contained just below the top of the stack.

With the invention's compiler, we decided to generate a set of about 70 (exactly 69) elementary instructions of the same type as the above three instructions. About 70 instructions are therefore generated and must be interpreted. The empirical choice of this number was based on the following considerations. If the compiler does a lot of things, the complexity of the various instructions to be executed by the interpreter, to produce the machine language instructions, will decrease. The memory space occupied by the interpreter will decrease (slightly). The overall result will be less interesting. In the opposite case, the interpreter itself will occupy a prohibitively large space in memory. The power of the intermediate language is therefore determined empirically. It is the result of lessons learned from both successful and failed experiences. If one has a very powerful intermediate language, one will automatically have a larger set of instructions in this language. A larger set of instructions in the intermediate language will in turn result in a more voluminous interpreter. By a powerful compiler, we mean a compiler that is capable of handling 3 or 4 stack levels, including adding and multiplying them. It could be designed to do many more things, but this would result in a much more powerful compiler and intermediate language, requiring a much longer interpreter since the number of instructions to be translated into actual machine language would be much larger. Therefore, the gain in memory space that would be obtained by trying to use an intermediate language would be lost because of a very long interpreter. A very powerful intermediate language would have instructions that are similar to those of the higher-level language. If the interpreter does all the work, it is going to be long and occupy a lot of memory space.

Once the intermediate program has been created, it is stored in section 5 of memory 6.

Then, in Step 17, or at an earlier stage, a command-interpretation program is written; the

program is specific to the type of microprocessor being used in the card. For the microprocessor used in the example, the command-interpretation program is given in Attachment A. It has the following characteristics. It is written in the machine language of the microprocessor (even though, in the attachment, it is given in its assembly form). The interpreter takes as input the expressions generated by the compiler and converts them into instructions that can be directly understood by the microprocessor. Its main algorithm is simple. The first expression to be encountered is executed and the program's pointer moves to the next expression. For I=14, three instructions were generated: 1) put the address of I on the stack; 2) put the number 14 on top of I's address in the stack; and 3) assign the value located at the top of the stack to the address contained below it in the stack.

The interpreter takes the first instruction which in this case is "STACK address of variable I". Stacking I's address consists of placing it on a stack. The stacking instruction is coded in machine language on three bytes. The first byte is the code corresponding to the STACKing operation. The interpreter's program is such that when "STACK" is encountered, the next two bytes will contain the address of the place where the quantity to be stacked can be found, and the microprocessor will know that the address should be placed on the stack. For the microprocessor mentioned above, this gives an instruction called DJSR STACK. Its code, which can be found in Attachment A, is

PUSH_BASE_DS.

It consists of three instructions in machine language: these instructions are presented here in assembly language

```
move b7, b4
jsr stack and
jmp decoding
```

The command-interpretation program, which is directly executable by microprocessor 7, is then loaded (in Step 18) into section 13 of memory 6.

For the execution of the application program, the invention works as follows. Each instruction of the intermediate program (i.e. the program stored in section 5 of memory 6) is considered as a macro-instruction which is decoded in Step 19 using the interpreter. This macro-instruction is equivalent to a series of micro-instructions which are directly executable by the microprocessor. The micro-instructions in this series are executed one after the other until the last one. Once the last micro-instruction of the series has been executed, the program returns to the decoding of the next macro-instruction of the intermediate program.

Figure 3 explains how the invention works. It summarizes the same elements that have been described so far. This figure also includes a program counter 22 which is used for the sequential processing of the IM instructions of the intermediate program stored in memory 5. An instruction of this intermediate program will for instance be coded on three bytes: one byte containing an instruction code and two bytes containing the address of an operand. The instruction code stored in RAM memory 9 is first sent to an instruction decoder 23 of microprocessor 7. The interpreter recognizes the instruction code IM using the ALU 24 and the instruction decoder 23, because of the presence of the control signals corresponding to this stage of program execution. This instruction is recognized as being an instruction in the intermediate language. Once this instruction has been decoded, the series 27 of micro-instructions IP to IP + 4, corresponding to the decoding of instruction IM, will be loaded in memory 9. The loading of this series is done by sending instruction IM to the decoder 23, then to the ALU 24, and finally to an addressing array 241. This array can also be replaced with a small subroutine to perform the same task. The interpreter thus contains a certain number (69) of micro-instruction series, for instance the series 27 through 30. Another program

counter is used to execute instructions IP through IP+4. These instructions act on operand 25 contained in data storage 8, whose address has been decoded by address register 26. One by one, these instructions IP to IP+4 are passed through the instruction decoder 23 before being sent to unit 24 where they act on operand 25. We remind the reader that instructions IP to IP+4 are directly executable by unit 24. The end of a series of micro-instructions is marked by the presence of an end micro-instruction whose function is to trigger the control signals which are necessary to go to the next intermediate-language instruction. Such a stack of micro-instructions can be easily handled by microprocessor 7 which has an integrated stack management feature. It can also be simulated by those microprocessors which do not have such a direct stack management feature. It is preferable to have this stack management feature in the operating system of microprocessor 7.

For the development of encryption algorithms, for instance in the case of banking applications, it may be preferable to use an assembler because of shorter execution times. However, this does not exclude the use of the invention; during the execution of the program, one would only need to call subroutines written in machine language (after ASSEMBLY). These subroutines can also be loaded in section 13 of memory 6.

CLAIMS

1 - Portable microcircuit support medium in which the microcircuit contains a microprocessor, a program storage (ROM), a data storage (EEPROM), and means to have the microprocessor execute a program stored in the program storage; this medium is characterized by the fact that the program storage contains a section where an interpretation program, specific to the type of microprocessor being used, is stored; the function of this program is to make the microprocessor execute, one by one, the instructions of an intermediate application program stored either in the program storage or in the data storage, after these instructions had been individually interpreted by the interpretation program, so that, for instance, the intermediate application program can act on data contained in the data storage.

2 - Support medium based on Claim 1 and characterized by the fact that the interpretation program is capable of translating an intermediate application program compiled from a program written in any one of the following programming languages:

C
PASCAL
COBOL
BASIC
ADA
FORTRAN

3 - Support medium based on Claim 1 or Claim 2, and characterized by the fact that program storage has also a section which contains the microprocessor's operating system.

4 - Support medium based on any one of Claims 1 through 3 and characterized by the fact that the microcircuit contains means for creating, on a temporary basis, the instructions to be implemented by the microprocessor, and a working memory to store these instructions during their short-lived existence.

5 - Process for using a microcircuit support medium where the microcircuit contains a microprocessor, a program storage (ROM), a data storage (EEPROM), and means (RAM) to make the microprocessor execute a program contained in either the program or data storage; this process is characterized by the fact that it consists of the following steps:

- a command-interpretation program is loaded in the program storage,
- an intermediate application program is loaded in the data storage,
- at least one instruction of the intermediate program is interpreted by the command-interpretation program,
- the microprocessor executes the intermediate instruction that has been decoded
- the same procedure is repeated for the next instruction of the intermediate program.

Plate 1/8

EEPROM

ROM/P

ROM/I

RAM

μ P

I/O

FIGURE 1

Plate 2/8
FIGURE 2

14: Writing of an application program in a higher-level language

15: Intermediate compilation

16: Storage of the intermediate program in EEPROM memory

17: Writing of a command-interpretation program

18: Storage of the interpreter in a (read-only) memory section of the microcircuit

19: Interpretation of the intermediate program instructions

20: Execution of executable instructions

END

Plate 3/8

6: ROM

Début = Start

Fin = End

9: RAM

22: PROGRAM COUNTER

Instruction (Byte 1)

Address (Byte 2)

Address (Byte 3)

XXXX

8 and 25: EEPROM DATA STORAGE

23: INSTRUCTION DECODER

26: ADDRESS REGISTER

24: ALU

241: Array of addresses

Towards machine 3

FIGURE 3

Pl. 4/8

* Pages 4-8 are code + haven't been translated. These pages are attached from FR patent 2667171

INTERPRETEUR_C_CARD, pour ASSEMBLEUR 6805

20 Janvier 1990

VERSION 1.0

```

a1 equ    $40
a2 equ    $100
a3 equ    $09
a4 equ    $00
move      macro r,d
    lda    d+1
    sta    r+1
    lda    d
    sta    r
endm
addw      macro r,d1,d2
    lda    d1+1
    add    d2+1
    sta    r+1
    lda    d1
    adc    d2
    sta    r
endm
subw      macro r,d1,d2
    lda    d1+1
    sub    d2+1
    sta    r+1
    lda    d1
    sbc    d2
    sta    r
endm

.PAGE0
org       a1
a5        rmb    2
b         rmb    2
b1        rmb    2
b2        rmb    2
b3        rmb    1
b4        equ    *
b5        rmb    1
b6        rmb    1
b7        rmb    2
b8        rmb    1
b9        rmb    1
c1        rmb    2
rts       rmb    1
.CODE
org       a2
jmp       ini
cbw       equ    *
    lda    b5
    clr    b5
    sta    b6
    bpl    cbw1
    dec    b5
cbw1      equ    *
rts
store_word equ *
    lda    $5c7
    sta    b9
    lda    b7
    jsr    b9
    inc    c1+1
    bne    store_word1
    inc    c1
store_word1 equ *
    lda    b7+1
    jsr    b9
    rts
store_byte equ *
    lda    $5c7
    sta    b9
    lda    b5
    jsr    b9
    rts
load_word equ *
    lda    $5c6
    sta    b9
    jsr    b9
    sta    b7
    inc    c1+1
    bne    load_word1
    inc    c1
load_word1 equ *
    jsr    b9
    sta    b7+1
    rts
load_byte equ *
    lda    $5c6
    sta    b9
    jsr    b9
    rts
empiler   equ    *
    ldx    b+1
    decx
    decx
    stx    b+1
    stx    c1+1
    clr    c1
    lda    b7
    sta    0,x
    lda    b7+1
    sta    1,x
    rts
depiler   equ    *
    ldx    b+1
    lda    0,x
    sta    b7
    lda    1,x
    sta    b7+1
    incx
    incx
    stx    b+1
    rts
depiltwo  equ    *
    jsr    depiler
    move    b4,b7
    jsr    depiler
    rts
fetch     equ    *
    move    c1,a5
    lda    $5c6
    sta    b9
    jsr    b9
    inc    a5+1
    bne    fetch1
    inc    a5
fetch1    equ    *
    rts
cmpw      equ    *
    lda    b7+1
    sub    b4+1
    sta    b4+1
    bpl    cmpw1
    lda    b7
    sbc    b4
    bmi    cmpw2
    ora    b4+1
    and    $57f
    bra    cmpw3
cmpw1     lda    b7
    sbc    b4
cmpw2     ora    b4+1
cmpw3     rts
ini       equ    *
    lda    $581
    sta    b9+3
    lda    $a3
    sta    a5
    lda    $a4
    sta    a5+1
    jsr    fetch
    sta    b3
    stop
    swi
    cmp    $570
    blo    aiguillage1
    jsr    fetch
    sta    b5
    lda    b3
    cmp    $580
    blo    aiguillage2
    jsr    fetch
    sta    b6
    lda    b3
    and    $57f
    tax
    aslx
    lda    tabcod3,x
    sta    c1
    lda    tabcod3+1,x
    bra    aiguillagefinal
aiguillage1 equ *
    ldx    b3
    aslx
    lda    tabcod1,x
    sta    c1
    lda    tabcod1+1,x
aiguillagefinal equ *
    sta    c1+1
    lda    $5cc
    sta    b9
    jmp    b9
aiguillage2 equ *
    lda    b3
    and    $58f
    tax
    aslx
    lda    tabcod2,x
    sta    c1
    lda    tabcod2+1,x
    bra    aiguillagefinal

```

Pl. 5/8

```

*
spush_immediat equ *
spush_base_ds equ *
    lda b5
    sta b7+1
    clr b7
    jsr empiler
    jmp decodage
debug equ *
    jsr depiler
    lda b7
    ldx b7+1
    swi
    jmp decodage
sval_word_bp equ *
    jsr cbw
    addw c1,b1,b4
    jsr load_word
    jsr empiler
    jmp decodage
spush_base_bp equ *
    jsr cbw
    addw b7,b1,b4
    jsr empiler
    jmp decodage
cmpvrai equ *
    lda #1
    bra fin_comparaison
cmpfaux equ *
    clra
fin_comparaison equ *
    sta b7+1
    clr b7
    jsr empiler
    jmp decodage
ega equ *
    jsr depiltwo
    jsr cmpw
    beq egavrai
    jmp cmpfaux
egavrai equ *
    jmp cmpvrai
dif equ *
    jsr depiltwo
    jsr cmpw
    bne difvrai
    jmp cmpfaux
difvrai equ *
    jmp cmpvrai
inf equ *
    jsr depiltwo
    jsr cmpw
    bmi infvrai
    jmp cmpfaux
infvrai equ *
    jmp cmpvrai
sup equ *
    jsr depiltwo
    jsr cmpw
    bhi supvrai
    jmp cmpfaux
supvrai equ *
    jmp cmpvrai
inf_ega equ *
    jsr depiltwo
    jsr cmpw
    bmi infegavrai
    beq infegavrai
    jmp cmpfaux
infegavrai equ *
    jmp cmpvrai

sup_ega equ *
    jsr depiltwo
    jsr cmpw
    bhi supegavrai
    beq supegavrai
    jmp cmpfaux
supegavrai equ *
    jmp cmpvrai
sadd_sp equ *
    lda b+1
    add b5
    sta b+1
    lda b
    add #0
    sta b
    jmp decodage
ret_fon equ *
    move b,b1
    jsr depiler
    move b1,b7
    jsr depiler
    move a5,b7
    jmp decodage
sval_byte_bp equ *
    jsr cbw
    addw c1,b1,b4
empiler_byte equ *
    jsr load_byte
    sta b7+1
    clr b7
    jsr empiler
    jmp decodage
val_byte equ *
    jsr depiler
val_bytel equ *
    move c1,b7
    jmp empiler_byte
val_byte_ds equ *
    move c1,b4
    jmp empiler_byte
val_byte_bp equ *
    addw c1,b1,b4
    jmp empiler_byte
sval_byte_ds equ *
    lda b5
    sta c1+1
    clr c1
    jmp empiler_byte
dup_stackb equ *
    jsr depiler
    jsr empiler
    jmp val_bytel
sto_byte equ *
    jsr depiler
    lda b7+1
    sta b5
    jsr depiler
    move c1,b7
    jsr store_byte
    jmp decodage
deb_fon equ *
    move b7,b1
    jsr empiler
    move b1,b
    jmp decodage

intrsys equ *
    move c1,b
    jsr load_word
    lda #6
    cmp b7+1
    bne code7
    lda b+1
    add #2
    sta c1+1
    lda b
    adc #0
    sta c1
    jsr load_word
    lda b7+1
    nop
    jmp decodage
code7 equ *
    lda #7
    cmp b7+1
    bne code8
    rti
    jmp decodage
code8 equ *
    lda #8
    cmp b7+1
    bne code9
    bil code81
    jmp decodage
code9 equ *
    jmp decodage
val_word equ *
    jsr depiler
    move c1,b7
    jsr load_word
    jsr empiler
    jmp decodage
dup_stackw equ *
    jsr depiler
    jsr empiler
    move c1,b7
    jsr load_word
    jsr empiler
    jmp decodage
val_word_bp equ *
    addw c1,b1,b4
val_wordl equ *
    jsr load_word
    jsr empiler
    jmp decodage
val_word_ds equ *
    move c1,b4
    jmp val_wordl
sval_word_ds equ *
    lda b5
    sta c1+1
    clr c1
    jmp val_wordl
sto_word equ *
    jsr depiler
    move b4,b7
    jsr depiler
    move c1,b7
    move b7,b4
    jsr store_word
    jmp decodage
or_logical equ *
    jsr depiltwo
    tst b4+1
    beq or_logical1
    jmp cmpvrai

```

Pl. 6/8

```

or_logical1 equ *
    tst b7+1
    beq or_logical2
    jmp cmpvrai
or_logical2 equ *
    jmp cmpfaux
and_logical equ *
    jsr depiltwo
    tst b4+1
    beq and_logical1
    tst b7+1
    beq and_logical1
    jmp cmpvrai
and_logical1 equ *
    jmp cmpfaux
not_logical equ *
    jsr depiler
    tst b7+1
    beq not_logical1
    jmp cmpfaux
not_logical1 equ *
    jmp cmpvrai
mul equ *
    jsr depiltwo
    move c1,b4
    ldx #16
    clr b4
    clr b4+1
    ror b7
    ror b7+1
mul1
    bcc mul2
    lda b4+1
    add c1+1
    sta b4+1
    lda b4
    adc c1
    sta b4
mul2
    ror b4
    ror b4+1
    ror b7
    ror b7+1
    decx
    bne mul1
    jsr empiler
    jmp decodage
mod equ *
    jsr depiltwo
    jsr div16
    jsr rmod
    jsr empiler
    jmp decodage
add equ *
indice_byte equ *
    jsr depiltwo
    addw b7,b7,b4
    jsr empiler
    jmp decodage
sub equ *
    jsr depiltwo
    subw b7,b7,b4
    jsr empiler
    jmp decodage
div equ *
    jsr depiltwo
    jsr div16
    jsr rdiv
    jsr empiler
    jmp decodage
shr equ *
    jsr depiltwo
    ldx b4+1
shrl
    lsr b7
    ror b7+1
    decx
    bne shrl
    jsr empiler
    jmp decodage
shl equ *
    jsr depiltwo
    ldx b4+1
shll
    lsl b7+1
    rol b7
    decx
    bne shll
    jsr empiler
    jmp decodage
and equ *
    jsr depiltwo
    lda b7+1
    and b4+1
    sta b7+1
    lda b7
    and b4
    sta b7
    jsr empiler
    jmp decodage
or equ *
    jsr depiltwo
    lda b7+1
    ora b4+1
    sta b7+1
    lda b7
    ora b4
    sta b7
    jsr empiler
    jmp decodage
xor equ *
    jsr depiltwo
    lda b7+1
    eor b4+1
    sta b7+1
    lda b7
    eor b4
    sta b7
    jsr empiler
    jmp decodage
neg equ *
    jsr depiler
    lda #0
    sub b7+1
    sta b7+1
    lda #0
    sbc b7
    sta b7
    jsr empiler
    jmp decodage
not equ *
    jsr depiler
    com b7+1
    com b7
    jsr empiler
    jmp decodage
inc_byte equ *
    jsr depiler
    move c1,b7
    jsr load_byte
    inca
    sta b5
    jsr store_byte
    jmp decodage
dec_byte equ *
    jsr depiler
    move c1,b7
    jsr load_byte
    deca
    sta b5
    jsr store_byte
    jmp decodage
inc_word equ *
    jsr depiler
    move c1,b7
    move b4,b7
    jsr load_word
    inc b7+1
    bne inc_word1
    inc b7
inc_word1 equ *
    move c1,b4
    jsr store_word
    jmp decodage
dec_word equ *
    jsr depiler
    move c1,b7
    move b4,b7
    jsr load_word
    lda b7+1
    sub #501
    sta b7+1
    lda b7
    sbc #00
    sta b7
dec_word1 equ *
    move c1,b4
    jsr store_word
    jmp decodage
indice_word equ *
    jsr depiler
    asl b7+1
    rol b7
    move b4,b7
    jsr depiler
    addw b7,b7,b4
    jsr empiler
    jmp decodage
push_ax equ *
    move b7,b2
    jsr empiler
    jmp decodage
pop_ax equ *
    jsr depiler
    move b2,b7
    jmp decodage
deb_fon_alloc equ *
    move b7,b1
    jsr empiler
    move b1,b
    subw b,b,b4
    jmp decodage
sdeb_fon_alloc equ *
    move b7,b1
    jsr empiler
    move b1,b
    lda b+1
    sub b5
    sta b+1
    lda b
    sbc #0
    sta b
    jmp decodage

```

Pl. 7/8

```

push_base_bp equ *
    addw b7,b1,b4
    jsr empiler
    jmp decodage
push_base_cs equ *
    addw b7,a5,b4
    jsr empiler
    jmp decodage
spush_base_cs equ *
    lda a5+1
    add b5
    sta b7+1
    lda a5
    adc #0
    sta b7
    jsr empiler
    jmp decodage
push_immediat equ *
push_base_ds equ *
    move b7,b4
    jsr empiler
    jmp decodage
dup_stack equ *
    jsr depiler
    jsr empiler
    jsr empiler
    jmp decodage
debut equ *
    clr b+1
    clr b
    clr b1
    clr b1+1
    jmp decodage
add_sp equ *
    addw b,b,b4
    jmp decodage
jmp equ *
jmp1 equ *
    addw a5,a5,b4
    jmp decodage
sjmp equ *
    jsr cbw
    jmp jmp1
jcf equ *
    jsr depiler
    tst b7+1
    beq jmp1
    jmp decodage
sjcf equ *
    jsr depiler
    tst b7+1
    beq sjmp
    jmp decodage
jcv equ *
    jsr depiler
    tst b7+1
    bne jmp1
    jmp decodage
sjcv equ *
    jsr depiler
    tst b7+1
    bne sjmp
    jmp decodage
call equ *
    move b7,a5
    jsr empiler
    addw a5,a5,b4
    jmp decodage
scall equ *
    move b7,a5
    jsr empiler
    jsr cbw
    addw a5,a5,b4
    jmp decodage
fin equ *
    wait
div16 equ *
    clr b8
    clrx
    clr c1
    clr c1+1
    incx
div161 equ *
    lsl b7+1
    rol b7
    rol c1
    rol c1+1
    lda c1
    sub b4+1
    sta c1
    lda c1+1
    sbc b4
    sta c1+1
    bcc div162
    lda b4+1
    add c1
    sta c1
    lda b4
    adc c1+1
    sta c1+1
    sec
div162 equ *
    rolx
    rol b8
    bcc div161
    rts
rdiv equ *
    comx
    stx b7+1
    ldx b8
    comx
    stx b7
    rts
rmod equ *
    ldx c1+1
    stx b7
    lda c1
    sta b7+1
    rts
tabcod1 equ *
    dw 0
    dw debut 1
    dw fin 2
    dw deb_fon 3
    dw ret_fon 4
    dw sto_byte 5
    dw sto_word 6
    dw inc_byte 7
    dw inc_word 8
    dw dec_byte 9
    dw dec_word 0x0a
    dw val_byte 0x0b
    dw indice_byte 0x0c
    dw indice_word 0x0d
    dw and_logical 0x0e
    dw or_logical 0x0f
    dw or 0x10
    dw xor 0x11
    dw and 0x12
    dw ega 0x13
    dw dif 0x14
    dw inf 0x15
    dw sup 0x16
    dw inf_ega 0x17
    dw sup_ega 0x18
    dw shr 0x19
    dw shl 0x1A
    dw add 0x1B
    dw sub 0x1C
    dw mul 0x1D
    dw div 0x1E
    dw mod 0x1F
    dw neg 0x20
    dw not_logical 0x21
    dw not 0x22
    dw val_word 0x23
    dw push_ax 0x24
    dw pop_ax 0x25
    dw dup_stack 0x26
    dw dup_stackb 0x27
    dw dup_stackw 0x28
    dw intrsys 0x29
    dw debug 0x2A
tabcod2 equ *
    dw sdeb_fon_alloc 0x70
    dw spush_immediat 0x71
    dw spush_base_bp 0x72

```

```

tabcod1 equ *
dw 0
dw debut 1
dw fin 2
dw deb_fon 3
dw ret_fon 4
dw sto_byte 5
dw sto_word 6
dw inc_byte 7
dw inc_word 8
dw dec_byte 9
dw dec_word 0x0a
dw val_byte 0x0b
dw indice_byte 0x0c
dw indice_word 0x0d
dw and_logical 0x0e
dw or_logical 0x0f
dw or 0x10
dw xor 0x11
dw and 0x12
dw ega 0x13
dw dif 0x14
dw inf 0x15
dw sup 0x16
dw inf_ega 0x17
dw sup_ega 0x18
dw shr 0x19
dw shl 0x1A
dw add 0x1B
dw sub 0x1C
dw mul 0x1D
dw div 0x1E
dw mod 0x1F
dw neg 0x20
dw not_logical 0x21
dw not 0x22
dw val_word 0x23
dw push_ax 0x24
dw pop_ax 0x25
dw dup_stack 0x26
dw dup_stackb 0x27
dw dup_stackw 0x28
dw intrsys 0x29
dw debug 0x2A

```

```

*
tabcod2 equ *
dw sdeb_fon_alloc 0x70
dw spush_immediat 0x71
dw spush_base_bp 0x72
dw sjmp 0x73
dw scall 0x74
dw sadd_sp 0x75
dw spush_base_cs 0x76
dw spush_base_ds 0x77
dw sjcf 0x78
dw sval_word_bp 0x79
dw sval_word_ds 0x7a
dw sval_byte_bp 0x7b
dw sval_byte_ds 0x7c
dw sjcv 0x7d

```

```

*
tabcod3 equ *
dw deb_fon_alloc 0x8
dw push_immediat 0x8
dw push_base_bp 0x8
dw jmp 0x83
dw call 0x84
dw add_sp 0x85
dw push_base_cs 0x86
dw push_base_ds 0x87
dw jcf 0x88
dw val_word_bp 0x89
dw val_word_ds 0x8a
dw val_byte_bp 0x8b
dw val_byte_ds 0x8c
dw jcv 0x8d
end

```


This Page is inserted by IFW Indexing and Scanning
Operations and is not part of the Official Record

BEST AVAILABLE IMAGES

Defective images within this document are accurate representations of the original documents submitted by the applicant.

Defects in the images include but are not limited to the items checked:

- ☐ BLACK BORDERS
- ☐ IMAGE CUT OFF AT TOP, BOTTOM OR SIDES
- ☐ FADED TEXT OR DRAWING
- ☒ BLURED OR ILLEGIBLE TEXT OR DRAWING
- ☐ SKEWED/SLANTED IMAGES
- ☐ COLORED OR BLACK AND WHITE PHOTOGRAPHS
- ☐ GRAY SCALE DOCUMENTS
- ☐ LINES OR MARKS ON ORIGINAL DOCUMENT
- ☐ REPERENCE(S) OR EXHIBIT(S) SUBMITTED ARE POOR QUALITY
- ☐ OTHER: _____

IMAGES ARE BEST AVAILABLE COPY.

**As rescanning documents *will not* correct images
problems checked, please do not report the
problems to the IFW Image Problem Mailbox**